ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μελέτη Εφαρμογής Παράλληλων Μεθόδων σε Επερωτήματα Βάσεων Δεδομένων

Γιώργος Ινιάτης

Επιβλέπων Καθηγητής

Pedro Trancoso

Ατομική Διπλωματική Εργασία

Μελέτη Εφαρμογής Παράλληλων Μεθόδων σε Επερωτήματα Βάσεων Δεδομένων

Γιώργος Ινιάτης

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2013

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2013

Ευχαριστίες

Με την διεκπεραίωση της παρούσης διπλωματικής εργασίας, θα ήθελα να εκφράσω τις μεγάλες μου ευχαριστίες στον υπεύθυνο και επιβλέποντα καθηγητή μου κ. Pedro Trancoso, πρώτον για την ευκαιρία που μου έδωσε να ασχοληθώ με το συγκεκριμένο θέμα και δεύτερον για την υποστήριξη και την καθοδήγηση που μου προσέφερε όλο αυτό το διάστημα. Πιστεύω πώς υπο την δική του επίβλεψη, έχω αποκομίσει περισσότερες γνώσεις όσο αφορά την παράλληλη επεξεργασία αλλά κυρίως για το θέμα με το οποίο έχω ασχοληθεί.

Ιδιαίτερες ευχαριστίες θα ήθελα να εκφράσω και στους δύο διδακτορικούς φοιτητές, Αντρέα Διαβαστό και Κωνσταντίνο Χριστοφή, των οποίων οι συμβουλές ήταν πολύτιμες. Η εμπειρία τους και η βοήθεια τους έπαιξαν καθοριστικό ρόλο για την διεκπεραίωση της διπλωματικής μου εργασίας.

Τέλος, θερμές ευχαριστίες στον καθηγητή κ. Χρύση Γεωργίου, για την αποδοχή του στο να είναι ο 2^{ος} αξιολογητής της διπλωματικής μου εργασίας.

Περίληψη

Στη σημερινή εποχή, τα συστήματα Βάσεων Δεδομένων χρησιμοποιούνται, αν όχι σε όλους, σε πάρα πολλούς τομείς. Πολλές εφαρμογές – επερωτήματα (queries) Βάσεων Δεδομένων, έχουν μεγάλο έως και τεράστιο χρόνο εκτέλεσης στις περιπτώσεις όπου ο αριθμός των δεδομένων που έχουν ως είσοδο είναι πολύ μεγάλος. Πολλές τεχνικές έχουν χρησιμοποιηθεί για την βελτίωση τέτοιων εφαρμογών, όπως η τεχνική του κατακερματισμού, η σωστή σειρά με την οποία εκτελούνται οι λειτουργίες σύνδεσης (join operations) κ.α. Τα τελευταία όμως χρόνια, ιδιαίτερη έμφαση δίνεται στην βελτιστοποίηση της επίδοσης μεγάλων εφαρμογών Βάσεων Δεδομένων μέσω του παραλληλισμού που παρέχεται από τους πολυπύρηνους επεξεργαστές, αφού πλέον όλες οι εταιρείες κατασκευής και ανάπτυξης επεξεργαστών χρησιμοποιούν το πολυπύρηνο μοντέλο. Η νέα τάση βελτιστοποίησης των εφαρμογών που τρέχουν σε Βάσεις Δεδομένων, έχει ώς βασικό στόχο την εκμετάλλευση του παραλληλισμού που παρέχουν οι πλέον Παράλληλες Βάσεις Δεδομένων.

Στην παρούσα διπλωματική εργασία, μελετάμε την επίδοση που μπορούν να προσφέρουν σε εφαρμογές Βάσεων Δεδομένων (queries) οι διάφορες μορφές παραλληλισμού και κυρίως ο παραλληλισμός με διασωλήνωση. Προσπαθούμε δηλαδή, να βελτιώσουμε την επίδοση τέτοιων εφαρμογών μέσω του παραλληλισμού και να πετύχουμε αν όχι ιδεατή, πολύ ψηλή επιτάχυνση (speedup).

Οι δύο μορφές παραλληλισμού που χρησιμοποιούνται για την αύξηση της επίδοσης των εφαρμογών Βάσεων Δεδομένων, είναι ο παραλληλισμός δεδομένων (data parallelism) και ο παραλληλισμός με διασωλήνωση (pipeline parallelism). Στο τέλος της εργασίας αυτής, συγκρίνονται αυτές οι δύο μορφές παραλληλισμού βάσει της επίδοσης που προσφέρουν στις εφαρμογές που χρησιμοποιούνται για την αξιολόγησή τους.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

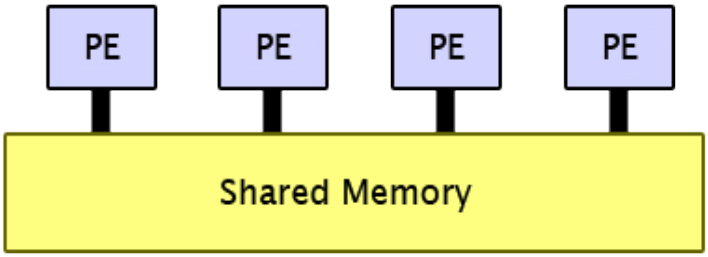
1.1 Παράλληλη Επεξεργασία και Αρχιτεκτονικές Επεξεργαστών

Η βελτίωση της επίδοσης των επεξεργαστών βασιζόταν αρχικά στην φιλοσοφία της αύξησης της συχνότητας ενός επεξεργαστή. Με την αύξηση της συχνότητας, ένας επεξεργαστής μπορεί να εκτελεί περισσότερους υπολογισμούς και πράξεις ανά μονάδα χρόνου. Η συνεχής όμως αυτή αύξηση της συχνότητας των επεξεργαστών, οδήγησε στο γνωστό “φράγμα ισχύος” (power wall), αφού αύξηση της συχνότητας συνεπάγεται περισσότερη κατανάλωση ενέργειας και αντίστοιχα αύξηση της θερμοκρασίας του συστήματος. Αυτό το πρόβλημα, οδήγησε στην εγκατάλειψη της προσπάθειας για ολοένα και περισσότερη αύξηση της συχνότητας ενός επεξεργαστή. Παρ’ όλα ταύτα, η βελτίωση της επίδοσης των επεξεργαστών δεν σταμάτησε εδώ, αφού λύση στο συγκεκριμένο πρόβλημα ήρθε να δώσει η ανάπτυξη των πολυπύρηνων επεξεργαστών. Συγκεκριμένα, οι εταιρείες παραγωγής επεξεργαστών επικεντρώνονται πλέον στην τοποθέτηση πολλών πανομοιότυπων ή διαφορετικών πυρήνων πάνω στον ίδιο επεξεργαστή, με πιο λίγη συχνότητα. Αυτό, εκτός του ότι επιτυγχάνει λιγότερη κατανάλωση ενέργειας, παρέχει επίσης την δυνατότητα της εκτέλεσης πολλών εργασιών στον επεξεργαστή παράλληλα, αφού ο επεξεργαστής αποτελείται από πολλές υπολογιστικές μονάδες.

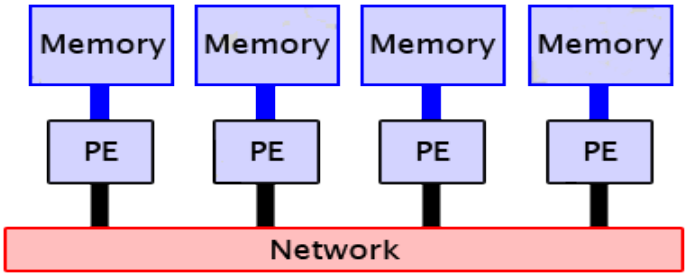
Η νέα αυτή τάση, αποτελεί σήμερα την βάση για την κατασκευή όλων των μοντέρνων επεξεργαστών. Επίσης, η τάση αυτή εφαρμόζεται για την κατασκευή των επεξεργαστών των έξυπνων κινητών συσκευών, οι οποίες μέρα με την μέρα εξελίσσονται και

προσπαθούν να φτάσουν το επίπεδο της υπολογιστικής ισχύος των σημερινών προσωπικών ηλεκτρονικών υπολογιστών, ενώ παράλληλα έχουν την ανάγκη για όσο το δυνατό χαμηλότερη κατανάλωση ενέργειας.

Οι νέες προκλήσεις που προκύπτουν με την ραγδαία ανάπτυξη της Παράλληλης Επεξεργασίας, αφορούν κυρίως θέματα μνήμης και την επικοινωνία μεταξύ των διαφόρων υπολογιστικών μονάδων. Οι πυρήνες ενός επεξεργαστή μπορούν είτε να διαμοιράζονται την ίδια μνήμη (μοντέλο κοινόχρηστης μνήμης), είτε έχουν ο καθένας την δική του προσωπική μνήμη (μοντέλο κατανεμημένης μνήμης). Στο μοντέλο κοινόχρηστης μνήμης οι πυρήνες επικοινωνούν μεταξύ τους μέσω κοινόχρηστων μεταβλητών, ενώ στο μοντέλο κατανεμημένης μνήμης η επικοινωνία μεταξύ των πυρήνων γίνεται με την χρήση ανταλλαγής μηνυμάτων. Υπάρχουν αρχιτεκτονικές οι οποίες συνδυάζουν τα δύο πιο πάνω μοντέλα.



Σχήμα 1.1: *Μοντέλο κοινόχρηστης μνήμης [17]*



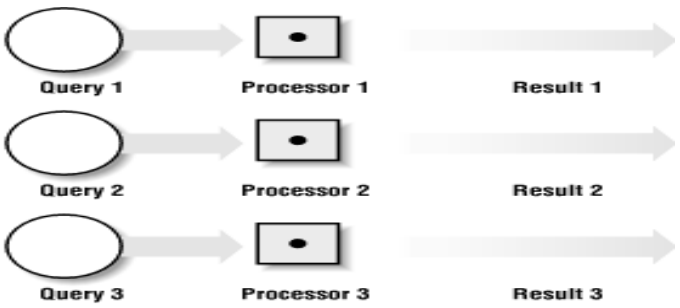
Σχήμα 1.2: *Μοντέλο κατανεμημένης μνήμης [18]*

1.2 Η σημαντικότητα της Χρήσης Παραλληλισμού στις Βάσεις Δεδομένων

Οι Βάσεις Δεδομένων γίνονται όλο και πιο μεγάλες αφού τα μεγέθη δεδομένων που αποθηκεύονται σε αυτές μετρούνται πλέον σε Terabytes και Petabytes. Ως φυσικό επόμενο, ο χρόνος εκτέλεσης ενός σειριακού επερωτήματος (Query) που χρησιμοποιεί μεγάλο όγκο δεδομένων είναι πάρα πολύ μεγάλος στις περισσότερες περιπτώσεις, όσες βελτιστοποιήσεις και αν γίνουν είτε από τον χρήστη είτε από τον μεταγλωτιστή. Για την βελτίωση της επίδοσης ενός τέτοιου επερωτήματος, είναι αναγκαία η χρήση του παραλληλισμού που προσφέρεται από τα Παράλληλα Συστήματα Βάσεων Δεδομένων [5, 6, 7]. Τα Παράλληλα Συστήματα Βάσεων Δεδομένων, αποτελούνται από πολλούς επεξεργαστές και πυρήνες, επιτρέποντας έτσι την αύξηση της επίδοσης των εφαρμογών Βάσεων Δεδομένων μέσω της χρήσης του παραλληλισμού.
Οι δύο τύποι παραλληλισμού που συναντιόνται σε Παράλληλες Βάσεις Δεδομένων, είναι οι εξής:

1) Παραλληλισμός ανάμεσα σε πολλά επερωτήματα (inter-query parallelism)
2) Παραλληλισμός ενός επερωτήματος (intra-query parallelism).

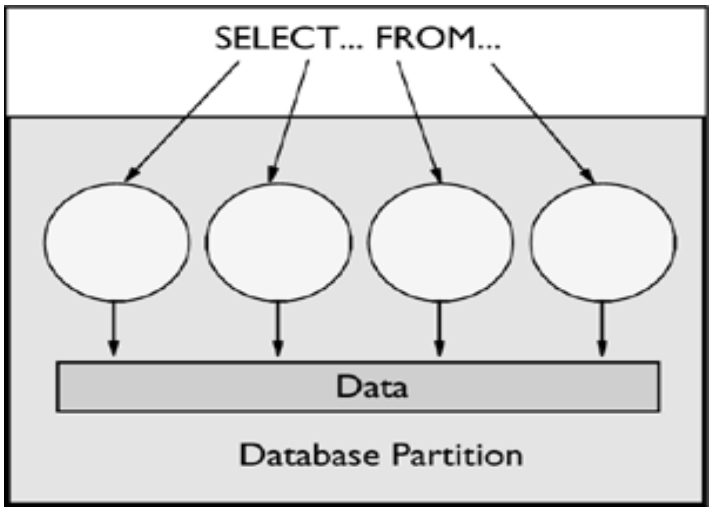
Ο πρώτος τύπος παραλληλισμού επιτρέπει την εκτέλεση πολλών επερωτημάτων την ίδια χρονική στιγμή και έχει ώς βασικό στόχο την αύξηση του ρυθμού εκτέλεσης των συγκεκριμένων επερωτημάτων.



Σχήμα 1.3: *Παραλληλισμός ανάμεσα σε πολλά επερωτήματα [19]*

Ο δεύτερος τύπος παραλληλισμού έχει ως βασικό στόχο την μείωση του χρόνου εκτέλεσης ενός συγκεκριμένου επερωτήματος. Η βασική ιδέα για την επίτευξη αυτού του στόχου, είναι να βρεθούν οι υπο-εργασίες του επερωτήματος οι οποίες μπορούν να εκτελεστούν παράλληλα και να μοιραστούν οι εργασίες αυτές μεταξύ των πυρήνων ώστε να βελτιστοποιηθεί η επίδοση όσο πιο πολύ γίνεται. Αυτό μπορεί να γίνει είτε με χρήση παραλληλισμού σε επίπεδο δεδομένων, είτε με τον παραλληλισμό διαφορετικών υπο-εργασιών του επερωτήματος, είτε με την χρήση διασωλήνωσης, είτε και με συνδυασμό των πιο πάνω [2].

Στην δική μου διπλωματική εργασία, χρησιμοποιείται ο δεύτερος τύπος παραλληλισμού για την βελτίωση της επίδοσης ξεχωριστών επερωτημάτων Βάσεων Δεδομένων. Αυτό προσπαθώ να το επιτεύξω χρησιμοποιώντας παραλληλισμό σε επίπεδο δεδομένων (data parallelism), καθώς επίσης και παραλληλισμό σε επίπεδο δεδομένων σε συνδυασμό με την τεχνική της διασωλήνωσης.



Σχήμα 1.4: *Παραλληλισμός ενός επερωτήματος [20]*

1.3 Κίνητρο

Στην σημερινή εποχή, οι εφαρμογές Βάσεων Δεδομένων χρησιμοποιούνται σχεδόν σε όλους τους τομείς, όπως παραδείγματος χάρη στην ιατρική όπου

φυλάγονται σε κάποια Βάση Δεδομένων τα στοιχεία και το ιστορικό των γιατρών και ασθενών. Επίσης, όπως έχει αναλυθεί και πιο πάνω, τα νέα υπολογιστικά συστήματα βασίζονται στην τεχνολογία και αρχιτεκτονική των πολυπύρηνων επεξεργαστών. Θα πρέπει λοιπόν να προσπαθήσουμε σε αυτή την διπλωματική εργασία, να βρούμε τρόπο να εκμεταλλευτούμε στο έπακρο τον παραλληλισμό που μας παρέχεται από κάποιο πολυπύρηνο επεξεργαστή, ούτως ώστε να επιτύχουμε, μέσω της χρήσης παραλληλισμού, όσο το δυνατό μεγαλύτερη επιτάχυνση στις εφαρμογές Βάσεων Δεδομένων που θα χρησιμοποιηθούν. Με τη διεκπεραίωση αυτού του έργου, συμβάλλουμε και εμείς με τη σειρά μας στην σοβαρή μελέτη που γίνεται από πολλά ερευνητικά εργαστήρια για την βελτίωση της επίδοσης εφαρμογών Βάσεων Δεδομένων σε πολυπύρηνο περιβάλλον.

1.4 Στόχος και Αναμενόμενα Αποτελέσματα

Η πειραματική αξιολόγηση των επερωτημάτων Βάσεων Δεδομένων που θα χρησιμοποιηθούν, θα γίνει σε δωδεκαπύρηνη μηχανή στην οποία μας έχει δώσει πρόσβαση το Τμήμα Πληροφορικής του Πανεπιστημίου Κύπρου. Ένας στόχος λοιπόν, είναι η εκμετάλλευση των 12 πυρήνων της μηχανής αυτής στο έπακρο, ώστε να επιτευχθεί ιδεατή επιτάχυνση, δηλαδή ίση με δώδεκα. Αυτό μπορεί να γίνει κατορθωτό με την χρήση καί των 12 πυρήνων καθ'όλη την διάρκεια εκτέλεσης των εφαρμογών που θα χρησιμοποιηθούν. Για την επίτευξη του στόχου αυτού, θα χρησιμοποιηθεί το μοντέλο των PThreads και η επικοινωνία μεταξύ των νημάτων θα γίνεται μέσω κοινόχρηστων μεταβλητών, αφού όλα τα νήματα θα διαμοιράζονται την ίδια μνήμη.

Θα προσπαθήσω λοιπόν, όπως έχει περιγραφεί στην προηγούμενη παράγραφο, την επίτευξη ιδεατής επιτάχυνσης στα πειράματα που θα εκτελεστούν μέσω του παραλληλισμού δεδομένων και της τεχνικής της διασωλήνωσης. Ο δεύτερος στόχος που έχουμε θέσει σε αυτή την εργασία, είναι η σύγκριση της επίδοσης που μας παρέχει ο παραλληλισμός με χρήση διασωλήνωσης, με την επίδοση που μας παρέχει ο παραλληλισμός δεδομένων.

Αυτό θα γίνει ώστε να δούμε ποια μορφή παραλληλισμού υστερεί έναντι της άλλης, για τις εφαρμογές που θα χρησιμοποιηθούν, ώστε στο τέλος να βγάλουμε ένα γενικό συμπέρασμα που θα αφορά το ποια μορφή παραλληλισμού εκ των δύο είναι πιο καλή για εφαρμογές με παρόμοια χαρακτηριστικά.

Στα πρώτα πειράματα, ο διαμοιρασμός του συνολικού φόρτου εργασίας των εφαρμογών μεταξύ των νημάτων γίνεται στατικά (static load balancing). Αυτό, μπορεί να έχει ώς αποτέλεσμα την μή επίτευξη ιδεατής επιτάχυνσης, στην περίπτωση που η εργασία κάποιων νημάτων είναι περισσότερη από την εργασία των υπολοίπων νημάτων. Λύση σε αυτό το πρόβλημα, μπορεί να δώσει η χρήση του δυναμικού καταμερισμού εργασίας (dynamic load balancing) μεταξύ των νημάτων. Στην παρούσα διπλωματική εργασία, έχει εφαρμοστεί και αυτή η προσέγγιση, ώστε να μπορέσουμε να βελτιώσουμε όσο πιο πολύ την επίδοση των εφαρμογών που χρησιμοποιούνται στην πειραματική αξιολόγηση.

Κάτι άλλο που πολύ πιθανό να παίξει σοβαρό ρόλο στην μη επίτευξη ιδεατής επιτάχυνσης στις εφαρμογές που χρησιμοποιούνται, είναι οι διάφορες στενώσεις όπως το διάβασμα ή γράψιμο από και πρός τον δίσκο (σε περίπτωση που το μέγεθος αρχείων που θα χρησιμοποιηθούν υπερβαίνει το μέγεθος της μνήμης RAM), ή η εξάρτηση μιάς εργασίας από κάποιαν άλλη εργασία η οποία δέν συμφέρει να εκτελεστεί παράλληλα.

Αναμένω λοιπόν με το τέλος αυτής της διπλωματικής εργασίας, την επίτευξη όσο το δυνατόν μεγαλύτερης επιτάχυνσης για τις εφαρμογές που θα χρησιμοποιηθούν, με την χρήση της διασωλήνωσης και του παραλληλισμού δεδομένων, αλλά και να αποκτήσω μια καθαρή εικόνα για το ποια μορφή παραλληλισμού είναι καλύτερη απο την άλλη για τις εφαρμογές που θα χρησιμοποιηθούν. Εκτός της μεγάλης επιτάχυνσης, πρέπει επίσης να διασφαλιστεί η ορθή λειτουργία των εφαρμογών μας, ούτως ώστε να μήν παράγουν λανθασμένα αποτελέσματα, κάτι το οποίο μπορεί να συμβεί σε περίπτωση που δεν υλοποιήσουμε σωστά τον παραλληλισμό και τις εξαρτήσεις μεταξύ των νημάτων.

1.5 Οργάνωση και Μεθοδολογία

Προτού αρχίσουμε την βελτιστοποίηση των επερωτημάτων Βάσεων Δεδομένων που έχουμε στην διάθεσή μας, έπρεπε να μελετήσουμε σχετικές δημοσιεύσεις οι οποίες έχουν σχέση με το θέμα μας. Ανάλυση και σύντομη περιγραφή των δημοσιεύσεων που έχω μελετήσει γίνεται στο επόμενο κεφάλαιο («Σχετική Δουλειά»).

Μετά από την μελέτη σχετικών δημοσιεύσεων και προτού αρχίσουμε την πειραματική διαδικασία, κατηγοριοποιήσαμε τα επερωτήματα Βάσεων Δεδομένων σε επερωτήματα χωρίς την χρήση της τεχνικής του κατακερματισμού και επερωτήματα που χρησιμοποιούν αυτή την τεχνική. Στη συνέχεια, μελετήσαμε το κάθε επερώτημα ξεχωριστά για να μπορούμε να έχουμε όσο το δυνατό καλύτερη αντίληψη της εργασίας που εκτελεί. Ακολούθως, χωρίσαμε την πειραματική διαδικασία για κάθε επερώτημα σε κάποια βήματα. Το πρώτο βήμα, ήταν η σειριακή εκτέλεση του συγκεκριμένου επερωτήματος. Το δεύτερο βήμα, ήταν η παράλληλη εκτέλεση του επερωτήματος με 12 νήματα (αφού τρέχουμε τα πειράματα σε δωδεκαπύρηνο επεξεργαστή), όπου ο παραλληλισμός γίνεται σε επίπεδο δεδομένων. Στο τρίτο βήμα, ξανα-εκτελούμε παράλληλα το επερώτημα με 12 νήματα, όμως με την διαφορά ότι ο παραλληλισμός γίνεται με συνδυασμό διασωλήνωσης (μοντέλο παραγωγού-καταναλωτή) και παραλληλισμού δεδομένων. Περιγραφή των συγκεκριμένων μορφών παραλληλισμού, αλλά και των υλοποιήσεών μας γίνεται στα επόμενα κεφάλαια. Εάν τα βήματα 2 και 3 δέν έχουν ώς αποτέλεσμα την αναμενόμενη ή ιδεατή επιτάχυνση, τότε επαναλαμβάνονται με συγκεκριμένες βελτιστοποιήσεις ώστε να έχουμε καλύτερα αποτελέσματα. Στην περίπτωση που και πάλι δεν έχουμε ψηλή επιτάχυνση, ψάχνουμε μέσω μετρικών εντός του κώδικα για να βρούμε που οφείλονται οι καθυστερήσεις που έχουν ώς αποτέλεσμα την μη αναμενόμενη επίδοση στις εφαρμογές.

1.6 Περιγραφή Δομής Ατομικής Διπλωματικής Εργασίας

Χωρίσαμε αυτή την διπλωματική εργασία σε 7 συνολικά κεφάλαια, κάθε ένα εκ των οποίων περιγράφει ένα μέρος της συνολικής εργασίας που έχουμε

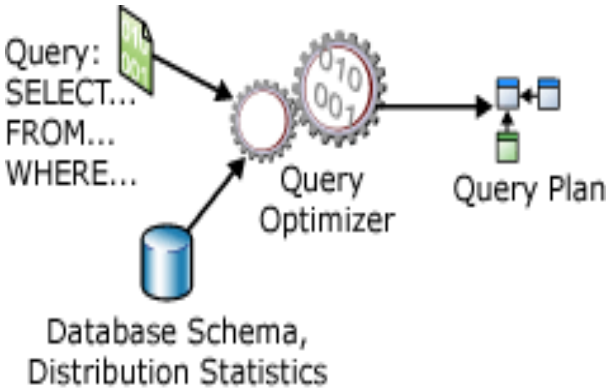
διεκπεραιώσει. Στο Κεφάλαιο 1, κάνουμε μία σύντομη εισαγωγή στην παράλληλη επεξεργασία και τις διάφορες αρχιτεκτονικές επεξεργαστών. Επίσης, εξηγούμε κάποια πράγματα που έχουν να κάνουν με την σημασία του παραλληλισμού για τις εφαρμογές Βάσεων Δεδομένων, ούτως ώστε να γίνει καλύτερη η κατανόηση επι του θέματος. Στο Κεφάλαιο 2 περιγράφεται η σχετική δουλειά που έχει γίνει για το θέμα με το οποίο θα ασχοληθούμε. Στο Κεφάλαιο 3, περιγράφουμε τις διάφορες μορφές παραλληλισμού, δίνοντας έμφαση στον παραλληλισμό με διασωλήνωση (και συγκεκριμένα του μοντέλου παραγωγού - καταναλωτή), αφού είναι η κύρια μορφή παραλληλισμού την οποία μελετούμε σε αυτή την διπλωματική εργασία. Στο Κεφάλαιο 4, περιγράφουμε τις εφαρμογές τις οποίες έχουμε χρησιμοποιήσει για την φάση της πειραματικής αξιολόγησης της εργασίας μας, καθώς επίσης και το σύστημα αξιολόγησης, δηλαδή την μηχανή στην οποία έγιναν όλα τα πειράματα. Στο Κεφάλαιο 5, περιγράφουμε τους αλγόριθμους στατικής και δυναμικής διασωλήνωσης που υλοποιήσαμε, καθώς επίσης και τις αντίστοιχες υλοποιήσεις με παραλληλισμό δεδομένων τις οποίες χρησιμοποιήσαμε για την σύγκριση των δύο μορφών παραλληλισμού. Στο Κεφάλαιο 6, περιγράφουμε την πειραματική αξιολόγηση των υλοποιήσεών μας μέσω γραφικών παραστάσεων και στο τέλος συγκρίνουμε τις δύο μορφές παραλληλισμού βάσει των αποτελεσμάτων μας. Τέλος, στο Κεφάλαιο 7 περιγράφουμε τα συμπεράσματα που έχουμε εξάξει μέσα απο την μελέτη του θέματος το οποίο αναλύουμε σε αυτή τη διπλωματική εργασία. Γίνεται επίσης μια σύντομη αναφορά στην μελλοντική εργασία που μπορεί να γίνει για την βελτίωση του έργου μας.

Κεφάλαιο 2

Σχετική Δουλειά

Προκειμένου να είμαστε έτοιμοι και σε θέση για να ξεκινήσουμε την υλοποίηση των στόχων μας, έπρεπε πρώτα να μελετήσουμε ορισμένες δημοσιεύσεις και παρουσιάσεις που έχουν σχέση με το θέμα μας, για να αποκομίσουμε όσο πιο πολλές γνώσεις που θα μας βοηθήσουν στην διεκπεραίωση του έργου μας. Πιο συγκεκριμένα, τα άρθρα που έχουμε μελετήσει αφορούν μελέτες για την βελτιστοποίηση επερωτημάτων (Queries) Βάσεων Δεδομένων με την χρήση παραλληλισμού.

Αρχικά, εστιάσαμε την μελέτη μας σε διάφορους βελτιστοποιητές επερωτημάτων, ούτως ώστε να μάθουμε για τις τεχνικές που χρησιμοποιούν για την βελτιστοποίηση των διαφόρων επερωτημάτων Βάσεων Δεδομένων.



Σχήμα 2.1: *Επεξεργασία ενός SQL Query και εξεύρεση βέλτιστου πλάνου μέσω ενός βελτιστοποιητή επερωτημάτων [21]*

Δύο τέτοιοι βελτιστοποιητές είναι ο HIVE [3] και ο AQUA [4]. Ο HIVE, μεταφράζει ένα επερώτημα σε ένα σέτ από “MapReduce” εργασίες, αλλά έχει το σοβαρό μειονέκτημα ότι ο προγραμματιστής πρέπει να δημιουργήσει ο ίδιος το βέλτιστο πλάνο εκτέλεσης του επερωτήματος προτού το δώσει ώς είσοδο στον

HIVE. Αυτό αποτελεί σοβαρό μειονέκτημα αφού η βελτιστοποίηση ενός επερωτήματος από μέρους του προγραμματιστή, εκτός του ότι είναι αρκετά δύσκολη μπορεί να είναι και χρονοβόρα. Από την άλλη, ο βελτιστοποιητής AQUA, αφού πάρει ώς είσοδο ένα επερώτημα, δημιουργά πρώτα ένα βέλτιστο πλάνο και στην συνέχεια το διασπά σε “MapReduce” εργασίες ούτως ώστε να εκτελεστεί παράλληλα επιτυγχάνοντας όσο το δυνατόν καλύτερη επίδοση. Μέσα από την μελέτη του συγκεκριμένου βελτιστοποιητή, έχω καταλάβει την σημαντικότητα της σειράς με την οποία ορίζουμε την εκτέλεση των λειτουργιών σύνδεσης ενός επερωτήματος.

Στην συνέχεια, απαραίτητη ήταν η μελέτη διαφόρων δημοσιεύσεων που παρουσιάζουν τεχνικές και αλγόριθμους που κάνουν χρήση της παράλληλης επεξεργασίας με σκοπό την βελτίωση του χρόνου εκτέλεσης ενός επερωτήματος. Μερικές απο αυτές τις μελέτες, υποστηρίζουν ότι ο παραλληλισμός ενός επερωτήματος με την τεχνική της διασωλήνωσης επιτρέπει περιορισμένο παραλληλισμό και συνεπώς όχι ιδεατή επιτάχυνση στην περίπτωση όπου ο αριθμός των εργασιών (tasks) αυτού του επερωτήματος είναι σχετικά μικρός [8]. Ένα παράδειγμα αλγορίθμου που κάνει χρήση παραλληλισμού με διασωλήνωση, είναι ο αλγόριθμος MTTC (Max-Throughput with Tree-Structured Precedence Constraints) [8], ο οποίος μεγιστοποιεί τον αριθμό των πλειάδων μίας σχέσης που μπορεί να τύχει επεξεργασίας σε κάποια μονάδα χρόνου. Άλλες τεχνικές, χρησιμοποιούν τον παραλληλισμό σε επίπεδο δεδομένων (Data Parallelism) για δεδομένα πολυδιάστατων πινάκων [9].

Επίσης, υπάρχει σχετική δουλειά η οποία μελετά την βελτίωση της επίδοσης των TPC-H εφαρμογών (queries) στον Many-Core Intel SCC [10]. Συγκεκριμένα, χρησιμοποιά τρείς διαφορετικούς αλγορίθμους για την εκτέλεση των επερωτημάτων. Ο πρώτος αλγόριθμος χρησιμοποιά παραλληλισμό σε επίπεδο δεδομένων με σειριακή εκτέλεση των λειτουργιών σάρωσης (scan operations), ο δεύτερος χρησιμοποιά την τεχνική του “nested-loop join”, ενώ ο τρίτος αλγόριθμος χρησιμοποιά την τεχνική της σύνδεσης με κατακερματισμό. Κανένας όμως απο αυτούς τους αλγόριθμους δεν κάνει χρήση διασωλήνωσης.

Για την διεκπεραίωση αυτής της διπλωματικής εργασίας, χρειάστηκε επίσης να μελετήσουμε την τεχνική του κατακερματισμού, προκειμένου να δοκιμάσουμε και αυτή την τεχνική στις εφαρμογές που θα χρησιμοποιήσουμε. Μελετήσαμε λοιπόν σχετική δουλειά η οποία περιγράφει αλγόριθμους που κάνουν χρήση αυτής της τεχνικής σε πολυπύρηνους επεξεργαστές [13]. Αυτοσκοπός μας δεν ήταν η υλοποίηση των συγκεκριμένων αλγορίθμων, αλλά να αποκομίσουμε τις απαραίτητες γνώσεις που θα χρησίμευαν για την δική μας υλοποίηση.

Μελετήσαμε επίσης το μοντέλο παραγωγού - καταναλωτή, αφού αποτελεί την βάση της δικής μας υλοποίησης. Αυτό έγινε μέσω μελέτης παρουσιάσεων που βρήκαμε στο διαδίκτυο, αλλά και μιας δημοσίευσης η οποία παρουσιάζει έναν προσομοιωτή, που δίνει την ευκαιρία στον χρήστη να μελετήσει το μοντέλο παραγωγού - καταναλωτή σε διαφορετικά πλαίσια [14].

Τέλος, μελετήσαμε και για τα χαρακτηριστικά που πρέπει να πληρούν οι εφαρμογές Βάσεων Δεδομένων για να μπορούν να χρησιμοποιούν τον παραλληλισμό με χρήση διασωλήνωσης [16]. Οι εφαρμογές πρέπει να αποτελούνται απο αλληλο-εξαρτώμενες εργασίες, δηλαδή απο εργασίες που για να ξεκινήσουν την εκτέλεσή τους χρειάζονται το αποτέλεσμα μιας άλλης ή περισσοτέρων εργασιών. Επίσης, για την εφαρμογή της διασωλήνωσης, πρέπει να υπάρχει και η ανάγκη της γρήγορης κατανάλωσης των ενδιάμεσων αποτελεσμάτων και συνεπώς του γρήγορου υπολογισμού του πρώτου αποτελέσματος, αλλά και του ρυθμού παραγωγής αποτελέσματος.

Όλα τα πιο πάνω, μας βοήθησαν να δημιουργήσουμε μια καλή γνώση για τα συστήματα Παράλληλων Βάσεων Δεδομένων και τις διάφορες τεχνικές που κάνουν χρήση του παραλληλισμού για την βελτιστοποίηση του χρόνου επεξεργασίας χρονοβόρων επερωτημάτων. Οι γνώσεις που αποκομίσαμε από αυτή την μελέτη, ήταν αυτές που μας βοήθησαν στην διεκπεραίωση αυτής της διπλωματικής εργασίας.

Η δική μας μελέτη, εστιάζεται στην μελέτη και αξιολόγηση του παραλληλισμού με χρήση διασωλήνωσης σε εφαρμογές Βάσεων Δεδομένων, αλλά παράλληλα

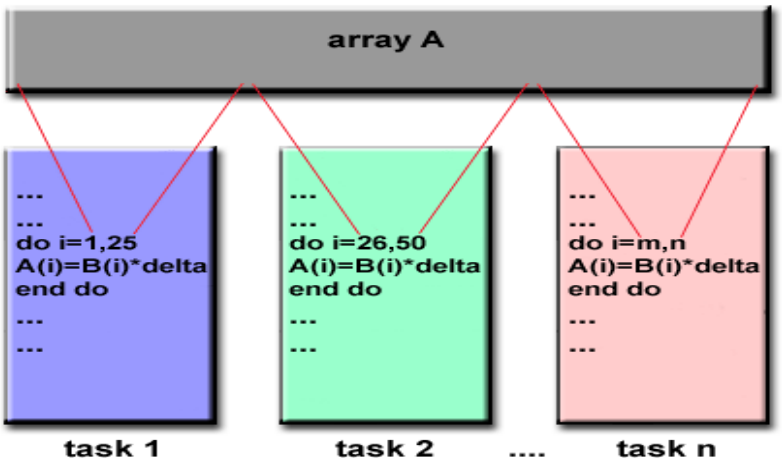
και στην σύγκρισή της με τον παραλληλισμό δεδομένων. Αυτή είναι και η κύρια διαφορά της δικής μας μελέτης απο τις άλλες σχετικές εργασίες που αναφέραμε πιο πάνω. Επίσης, μια άλλη διαφορά της δικής μας εργασίας απο τις άλλες εργασίες που έγιναν μέχρι στιγμής, είναι η μελέτη του αριθμού των νημάτων που δίνουμε στην κατηγορία των παραγωγών και στην κατηγορία των καταναλωτών, ανάλογα με το ποια κατηγορία έχει την περισσότερη συνολικά εργασία.

Κεφάλαιο 3

Μορφές Παραλληλισμού

3.1 Παραλληλισμός Δεδομένων

Ο Παραλληλισμός δεδομένων, εστιάζεται στον διαμοιρασμό των δεδομένων μίας συγκεκριμένης εργασίας-πράξης μεταξύ των υπολογιστικών μονάδων (πυρήνων) ενός συστήματος. Πιο συγκεκριμένα, όταν μια εφαρμογή εκτελεί ένα συγκεκριμένο σέτ από εντολές πάνω σε ένα σύνολο δεδομένων (SIMD – Single Instruction, Multiple Data), ο παραλληλισμός σε επίπεδο δεδομένων επιτυγχάνεται με το να αναθέσουμε σε κάθε νήμα το ίδιο σέτ εντολών σε διαφορετικά όμως δεδομένα.



Σχήμα 3.1: *Παράδειγμα εκτέλεσης της ίδιας πράξης από n-νήματα σε διαφορετικό σύνολο απο δεδομένα του ίδιου πίνακα [22]*

Αφού ολοκληρώσουν τα νήματα την εργασία που τους έχει ανατεθεί, τότε επικοινωνούν μεταξύ τους για τον υπολογισμό του τελικού αποτελέσματος. Ένα τέτοιο παράδειγμα, είναι ο υπολογισμός του αθροίσματος ενός μονοδιάστατου πίνακα. Κάθε νήμα αναλαμβάνει να υπολογίσει το άθροισμα ενός ξεχωριστού συνόλου δεδομένων του πίνακα. Αφού τα νήματα τελειώσουν με τον υπολογισμό του αθροίσματος των δεδομένων που τους αντιστοιχούν, γίνεται η πρόσθεση όλων των αθροισμάτων προκειμένου να υπολογιστεί το συνολικό άθροισμα του πίνακα.

3.2 Παραλληλισμός Εργασιών

Ο παραλληλισμός εργασιών, εστιάζεται στην εξεύρεση διαφορετικών εργασιών οι οποίες είναι ανεξάρτητες μεταξύ τους και στην παράλληλη εκτέλεσή τους. Αυτό σημαίνει, ότι μπορεί να εκτελούνται παράλληλα διαφορετικές εργασίες πάνω στο ίδιο σύνολο δεδομένων, είτε σε διαφορετικό σύνολο δεδομένων. Για να γίνει κάτι τέτοιο, πρέπει να ληφθούν υπόψιν οι οποιεσδήποτε εξαρτήσεις μεταξύ των διαφόρων εργασιών, ούτως ώστε να μήν εκτελούνται δύο εργασίες παράλληλα εάν η μία απο αυτές χρειάζεται το αποτέλεσμα της άλλης προτού ξεκινήσει την εκτέλεσή της.

Στην πιο κάτω εικόνα, παρατηρούμε ένα παράδειγμα όπου έχουμε παραλληλισμό μεταξύ δύο ανεξάρτητων συναρτήσεων-εργασιών. Το task_A εκτελείται από το νήμα 1, ενώ το task_B από το νήμα 2.

Thread 1:

```
task_A()
{
        int i;
        int array_A[10000];

        for(i = 0; i <10000; i++)
        {
                array_A[i] = 0;
        }
}
```

Thread 2:

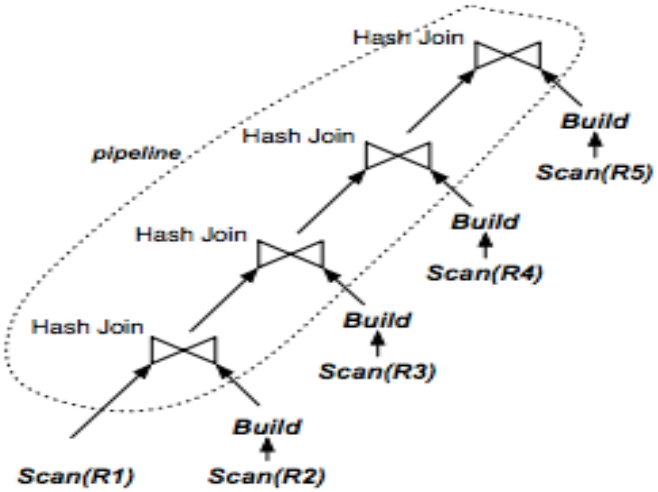
```
task_B()
{
        int i;
        int array_B[10000];

        for(i = 0; i <10000; i++)
        {
                array_B[i] = 1;
        }
}
```

Σχήμα 3.2: *Παράδειγμα εκτέλεσης διαφορετικών εργασιών την ίδια χρονική στιγμή*

3.3 Παραλληλισμός με Διασωλήνωση

Ο παραλληλισμός με διασωλήνωση, είναι ξεχωριστός τύπος της προηγούμενης μορφής παραλληλισμού που αναλύσαμε πιο πάνω (παραλληλισμός εργασιών). Η κύρια διαφορά της διασωλήνωσης, είναι ότι παρέχει το πλεονέκτημα της εκτέλεσης 2 ή περισσότερων εξαρτωμένων μεταξύ τους εργασιών παράλληλα, όταν μέρος του αποτελέσματος που παράγει μια εργασία μπορεί να χρησιμοποιηθεί από κάποια άλλη ή άλλες εργασίες άμεσα. Για παράδειγμα, έστω ότι έχουμε δύο εργασίες. Η μία εργασία δημιουργά 100 τυχαίους αριθμούς, ενώ η άλλη εργασία δέχεται ως είσοδο αυτούς τους αριθμούς και αφού υπολογίσει το άθροισμά τους (το οποίο έστω φυλάγεται στην μεταβλητή sum) το παρουσιάζει στην οθόνη.

Με την χρήση διασωλήνωσης στην συγκεκριμένη περίπτωση, μόλις η $1^{η}$ εργασία δημιουργήσει κάποιο αριθμό, διοχετεύεται αμέσως αυτός ο αριθμός στην $2^{η}$ εργασία η οποία τον προσθέτει στην μεταβλητή sum. Με αυτό τον τρόπο, και οι δύο εργασίες εκτελούνται παράλληλα, αφού η $2^{η}$ εργασία δέν χρειάζεται να περιμένει να λάβει καί τους 100 αριθμούς για να ξεκινήσει τον υπολογισμό του αθροίσματός τους. Αυτό είναι το μεγάλο πλεονέκτημα που παρέχεται από τον παραλληλισμό με την χρήση διασωλήνωσης. Στην πιο κάτω εικόνα έχουμε παραλληλισμό των εργασιών σύνδεσης ενός επερωτήματος με την χρήση διασωλήνωσης.

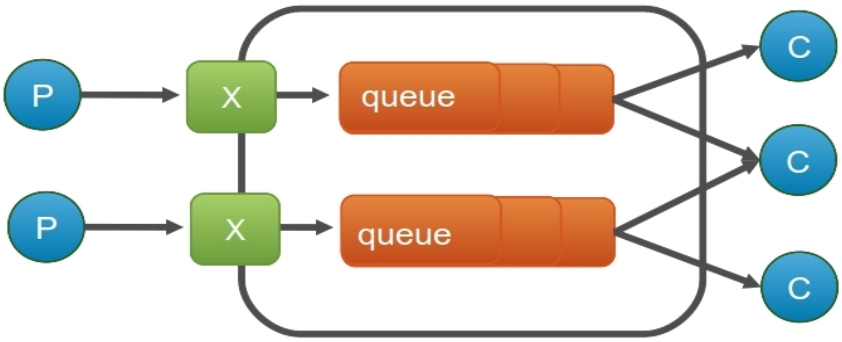


Σχήμα 3.3: *Παραλληλισμός των λειτουργιών "Hash - Join" ενός επερωτήματος με την χρήση ενός πλάνου διασωλήνωσης [8]*

Το αποτέλεσμα της σύνδεσης μεταξύ των σχέσεων R1 και R2 διοχετεύεται σε κάποιο άλλο νήμα το οποίο εκτελεί την σύνδεση για το αποτέλεσμα που έλαβε μαζί με την σχέση R3 κ.ο.κ.

3.3.1 Μοντέλο Παραγωγού - Καταναλωτή

Το μοντέλο παραγωγού-καταναλωτή είναι ειδική περίπτωση της τεχνικής της διασωλήνωσης. Στην παράλληλη επεξεργασία, ομαδοποιούμε τα νήματα σε παραγωγούς και καταναλωτές. Σε κάθε παραγωγό, αντιστοιχούν ένας ή περισσότεροι καταναλωτές. Παρομοίως, κάθε καταναλωτής αντιστοιχεί σε ένα ή περισσότερους παραγωγούς. Για να μπορέσει να λειτουργήσει αυτό το μοντέλο, πρέπει να υπάρχει κάποιος κοινός χώρος μνήμης (shared buffer) στον οποίο ο παραγωγός θα γράφει το αποτέλεσμα που παράγει, ενώ ο καταναλωτής θα διαβάζει από αυτό τον χώρο μνήμης και θα καταναλώνει το συγκεκριμένο αποτέλεσμα. Η υλοποίηση του συγκεκριμένου μοντέλου απαιτεί συγχρονισμό ούτως ώστε να μήν μπορεί να εισάγει δεδομένα ο παραγωγός στον κοινό χώρο μνήμης σε περίπτωση που αυτός είναι γεμάτος, αλλά και να αποφεύγεται παράλληλα το διάβασμα δεδομένων από τον καταναλωτή σε περίπτωση που ο κοινός χώρος μνήμης είναι άδειος. Επειδή σε αυτή την διπλωματική εργασία γίνεται χρήση αυτού του μοντέλου, ένα ζήτημα που πρέπει να μας απασχολήσει είναι ο αριθμός των νημάτων στα οποία θα αναθέσουμε ρόλο παραγωγού σε σχέση με τον αριθμό των νημάτων στα οποία θα αναθέσουμε ρόλο καταναλωτή. Αυτός είναι παράγοντας ο οποίος μπορεί να παίξει καθοριστικό ρόλο στην επίδοση των εφαρμογών μας.



Σχήμα 3.4: *Μοντέλο παραγωγού-καταναλωτή [23]*

3.4 Σύγκριση των Τριών Μορφών Παραλληλισμού

Οι τρείς μορφές παραλληλισμού τις οποίες έχουμε αναπτύξει πιο πάνω, έχουν τα δικά τους πλεονεκτήματα αλλά και τα δικά τους μειονεκτήματα. Ο παραλληλισμός δεδομένων μπορεί να προσφέρει αρκετά ψηλή επιτάχυνση σε μία εργασία κάποιας εφαρμογής, δεδομένου ότι ο διαμοιρασμός δεδομένων γίνεται κατάλληλα. Χρησιμοποιώντας όμως μόνο αυτή την μορφή παραλληλισμού, δεν μπορούμε να έχουμε εκτέλεση δύο ή περισσοτέρων εργασιών παράλληλα, κάτι το οποίο σε πολλές περιπτώσεις είναι επιθυμητό.

Ο παραλληλισμός εργασιών (task parallelism), παρέχει το πλεονέκτημα της εκτέλεσης ανεξάρτητων εργασιών την ίδια χρονική στιγμή. Το κυριότερο πρόβλημα όμως αυτής της μορφής παραλληλισμού, είναι το ότι σε πολλές εφαρμογές ο αριθμός των εργασιών είναι μικρότερος απο τον αριθμό των πυρήνων του επεξεργαστή, με αποτέλεσμα να μήν μπορεί να γίνει πλήρης εκμετάλλευση όλων των πυρήνων του επεξεργαστή (αφού κάθε ανεξάρτητη εργασία εκτελείται μόνο απο ένα νήμα).
Ένα καλό μοντέλο παραλληλισμού, είναι ο συνδυασμός του παραλληλισμού δεδομένων με τον παραλληλισμό ανεξάρτητων εργασιών. Δηλαδή, εκτέλεση πολλών εργασιών την ίδια χρονική στιγμή, με πολλά νήματα να εκτελούν μια εργασία μέσω παραλληλισμού δεδομένων για την συγκεκριμένη εργασία. Αυτό, επιτρέπει τον παραλληλισμό μίας εργασίας, καθώς επίσης και την παράλληλη εκτέλεση πολλών ανεξάρτητων εργασιών.

Ο παραλληλισμός με διασωλήνωση, χρησιμοποιείται για την παράλληλη εκτέλεση δύο ή περισσοτέρων αλληλο-εξαρτώμενων εργασιών. Αυτό έχει ώς αποτέλεσμα ψηλότερο throughput για το σύνολο των εργασιών, αφού μπορούν να εκτελούνται όλες οι εργασίες παράλληλα, με την μία εργασία να διοχετεύει το αποτέλεσμά της στην επόμενη κ.ο.κ. Αυτό έχει και ώς αποτέλεσμα την μείωση του χρόνου υπολογισμού για το πρώτο αποτέλεσμα, ενώ παράλληλα αυξάνει και τον ρυθμό υπολογισμού κάποιου αποτελέσματος. Για παράδειγμα, έστω ότι έχουμε 3 εργασίες, την εργασία Α, την εργασία Β και την εργασία Γ, όπου η Β εξαρτάται απο την Α και η Γ από την Β. Για να υπολογιστεί το πρώτο

αποτέλεσμα χωρίς διασωλήνωση, πρέπει να ολοκληρωθούν πρώτα οι εργασίες Α και Β. Με την χρήση διασωλήνωσης, δεν χρειάζεται να ολοκληρωθούν οι εργασίες Α και Β. Αρκεί απλά να παραχθεί κάποιο αποτέλεσμα απο τις εργασίες αυτές και να καταναλωθεί απο την εργασία Γ. Επίσης, ο ρυθμός υπολογισμού αποτελέσματος απο την εργασία Γ, είναι πολύ ψηλός αφού η εργασία Γ εκτελείται συνεχώς (παράλληλα με τις εργασίες Α και Β) κάτι που δεν συμβαίνει με τον παραλληλισμό δεδομένων. Αυτό είναι πολύ σημαντικό για διάφορες εφαρμογές, στις οποίες δεν μας ενδιαφέρει τόσο ο χρόνος εκτέλεσης όσο ο χρόνος υπολογισμού και παρουσίασης του πρώτου αποτελέσματος ή ο ρυθμός υπολογισμού του αποτελέσματος. Το κύριο μειονέκτημα όμως του παραλληλισμού με διασωλήνωση, είναι ότι δεν μειώνει τον χρόνο εκτέλεσης μίας συγκεκριμένης εργασίας, αφού κάθε εργασία (στην γνήσια διασωλήνωση) εκτελείται απο μόνο 1 νήμα. Αυτό το μειονέκτημα, μπορεί να αναιρεθεί μέσω του συνδυασμού του παραλληλισμού δεδομένων με διασωλήνωση. Δηλαδή, κάθε εργασία εκτελείται με παραλληλισμό δεδομένων, ενώ όλες οι αλληλο-εξαρτώμενες εργασίες εκτελούνται παράλληλα μέσω διασωλήνωσης.

Το συμπέρασμα στο οποίο καταλήγουμε, είναι ότι κάθε μορφή παραλληλισμού έχει κάποια πλεονεκτήματα και κάποια μειονεκτήματα. Ο συνδυασμός κάποιων μορφών παραλληλισμού, (όπως η διασωλήνωση μαζί με τον παραλληλισμό δεδομένων) συνδυάζει στην ουσία τα πλεονεκτήματα των μορφών αυτών και αναιρεί τα μειονεκτήματά τους. Σε αυτή την διπλωματική εργασία, η υλοποίηση μας που κάνει χρήση διασωλήνωσης, στην ουσία είναι συνδυασμός του παραλληλισμού με διασωλήνωση με τον παραλληλισμό δεδομένων και αυτό για να συνδυαστούν τα πλεονεκτήματα των δύο αυτών προσεγγίσεων.

Κεφάλαιο 4

Πειραματική Διάταξη

4.1 Περιγραφή Βασικών Λειτουργιών

Για την καλύτερη κατανόηση των εφαρμογών που έχουν χρησιμοποιηθεί, καλό είναι να γίνει μία σύντομη αναφορά στις λειτουργίες Βάσεων Δεδομένων που χρησιμοποιούνται εντός των εφαρμογών. Οι λειτουργίες αυτές είναι δύο [1]:

- Λειτουργία σάρωσης
- Λειτουργία συνδυασμού

Οι λειτουργίες σάρωσης, είναι στην ουσία λειτουργίες κατα τις οποίες διαβάζονται οι πλειάδες (records) ενός πίνακα, ώστε να επιλεγούν μόνο αυτές για τις οποίες ισχύει κάποια συνθήκη. Για να καταλάβουμε καλύτερα, έστω ότι σε μια Βάση Δεδομένων είναι αποθηκευμένος ένας πίνακας στον οποίο είναι καταχωρημένα τα στοιχεία όλων των φοιτητών του τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου. Μία πλειάδα αυτού του πίνακα, περιέχει όλα τα απαραίτητα στοιχεία για ένα φοιτητή Πληροφορικής, όπως το ονοματεπώνυμό του, το έτος εισδοχής του στο πανεπιστήμιο κ.τ.λ . Μια λειτουργία σάρωσης, θα μπορούσε να είναι η επιλογή των πλειάδων με τα στοιχεία των φοιτητών που ξεκίνησαν τις σπουδές τους στο Πανεπιστήμιο Κύπρου το 2008.

Η λειτουργία συνδυασμού είναι η λειτουργία η οποία συγκρίνει τις πλειάδες δύο ή περισσοτέρων πινάκων βάσει ενός κοινού χαρακτηριστικού αυτών των πινάκων και ακολούθως συνδυάζει (δηλαδή ενώνει) τις πλειάδες για τις οποίες ισχύει η σύγκριση. Συνήθως, το χαρακτηριστικό στο οποίο γίνεται η σύγκριση είναι το πρωτεύον κλειδί (primary key) για τον ένα πίνακα (το οποίο προσδιορίζει μοναδικά μία οντότητα – πλειάδα του πίνακα) και το ξένο κλειδί για τον δεύτερο πίνακα (ή τους υπόλοιπους πίνακες στην περίπτωση όπου η σύνδεση γίνεται σε περισσότερους απο 2 πίνακες). Για να καταλάβουμε και αυτή την λειτουργία καλύτερα, έστω ότι έχουμε τους πίνακες EMPLOYEE και DEPARTMENT (σχήμα 4.1). Στον πίνακα EMPLOYEE είναι καταχωρημένο το επίθετο κάθε εργαζομένου (LastName) και ο αριθμός του τμήματος (DepartmentID) στο οποίο δουλεύει. Στον πίνακα DEPARTMENT, είναι καταχωρημένος ο αριθμός (DepartmentID) και το όνομα (DepartmentName) για κάθε τμήμα. Ο αριθμός ενός τμήματος προσδιορίζει μοναδικά αυτό το τμήμα και συνεπώς αποτελεί το πρωτεύων κλειδί του πίνακα DEPARTMENT. Το χαρακτηριστικό DepartmentID του πίνακα EMPLOYEE, είναι ξένο κλειδί το οποίο αντιστοιχεί στο πρωτεύων κλειδί του πίνακα DEPARTMENT.

Employee table

LastName	DepartmentID
Rafferty	31
Jones	33
Steinberg	33
Robinson	34
Smith	34
John	NULL

Department table

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
35	Marketing

Σχήμα 4.1: *Οι πίνακες EMPLOYEE και DEPARTMENT*

Ένα επερώτημα το οποίο εκτελεί την σύνδεση των δύο αυτών πινάκων, θα μπορούσε να είναι το εξής:

```
SELECT *
FROM DEPARTMENT D, EMPLOYEE E
WHERE  D.DepartmentID = E.DepartmentID
```

Αυτό το επερώτημα, έχει ώς αποτέλεσμα την δημιουργία πλειάδων οι οποίες αποτελούνται από το επίθετο του εργαζομένου, τον αριθμό του τμήματος στο οποίο δουλεύει και το όνομα αυτού του τμήματος (π.χ Robinson, 34, Clerical).

Η σύνδεση μεταξύ δύο πινάκων, μπορεί να γίνει μέσω διάφορων τεχνικών. Δύο τέτοιες τεχνικές, είναι η τεχνική “nested-loop join” και η τεχνική “hash join”, τις οποίες χρησιμοποιήσαμε στις εφαρμογές μας για την υλοποίηση των λειτουργιών σύνδεσης.

Με την τεχνική ”nested-loop join” [1], το χαρακτηριστικό μιας πλειάδας ενός πίνακα, στο οποίο γίνεται η σύνδεση, συγκρίνεται με το αντίστοιχο χαρακτηριστικό όλων των πλειάδων του άλλου πίνακα. Η χρονική πολυπλοκότητα της σύνδεσης 2 πινάκων βάσει αυτής της τεχνικής είναι Θ(N * M), όπου N ο αριθμός των πλειάδων του ενός πίνακα και M ο αριθμός των πλειάδων του άλλου πίνακα.

Η τεχνική “hash-join”, χρησιμοποιά πίνακες κατακερματισμού για την λειτουργία σύνδεσης. Αυτό γίνεται ώς εξής: Αρχικά, ορίζεται ένας πίνακας κατακερματισμού για κάθε πίνακα απο τον οποίο θα διαβάσει η εφαρμογή [13]. Τα δεδομένα των πλειάδων ενός πίνακα (αυτό ισχύει για όλους τους πίνακες), καταχωρούνται στον πίνακα κατακερματισμού που αντιστοιχεί στον πίνακα αυτό βάσει της τιμής που ορίζεται στην συγκεκριμένη πλειάδα από τη συνάρτηση κατακερματισμού. Η τιμή που υπολογίζει και ορίζει η συνάρτηση αυτή για μια πλειάδα, εξαρτάται από την τιμή του χαρακτηριστικού στο οποίο θα γίνει η σύνδεση αυτού του πίνακα με κάποιον ή κάποιους άλλους πίνακες. Αυτό μειώνει σημαντικά τον χρόνο εκτέλεσης της σύνδεσης, αφού το χαρακτηριστικό (στο οποίο γίνεται η σύνδεση) μίας πλειάδας ενός πίνακα συγκρίνεται μόνο με χαρακτηριστικά πλειάδων ενός άλλου πίνακα τα οποία έχουν ίδια τιμή κατακερματισμού με αυτό. Συνεπώς, εάν έχουμε την σύνδεση μεταξύ δύο πινάκων, η χρονική πολυπλοκότητα με αυτή την τεχνική είναι Ω(N), O(N*M), όπου N και M ο αριθμός των πλειάδων των δύο πινάκων αντίστοιχα. Το ότι η χρονική πολυπλοκότητα είναι N στην καλύτερη περίπτωση και N*M στη χειρότερη, οφείλεται στο ότι η αναζήτηση σε ένα πίνακα κατακερματισμού έχει

χρονική πολυπλοκότητα ίση με 1 στην καλύτερη περίπτωση και M στην χειρότερη (όπου M ο αριθμός όλων των στοιχείων του πίνακα κατακερματισμού).

Βάσει της χρονικής πολυπλοκότητας των δύο τεχνικών, μπορούμε να παρατηρήσουμε ότι η τεχνική “hash join” υπερτερεί έναντι του “nested-loop join”. Πιο κάτω, παρουσιάζουμε τον πίνακα με την χρονική πολυπλοκότητα για τις δύο αυτές τεχνικές, όταν έχουμε σύνδεση μεταξύ δύο πινάκων.

	Nested-Loop Join	**Hash Join**
Χρονική Πολυπλοκότητα	Θ(N * M)	Ω(N), O(N * M)

Πίνακας 4.1: *Χρονική Πολυπλοκότητα για “nested-loop join” και “hash join“, όπου M και N ο αριθμός των πλειάδων των δύο πινάκων στους οποίους γίνεται η σύνδεση*

4.2 Περιγραφή Εφαρμογών

Οι δύο εφαρμογές που χρησιμοποιήσαμε για την αξιολόγηση της υλοποίησής μας, είναι τα επερωτήματα 3 και 12 από την σουίτα αξιολόγησης TPC-H. Χρησιμοποιήσαμε αυτές τις εφαρμογές, αφού μπορούν να εκμεταλλευτούν τον παραλληλισμό που τους προσφέρεται από το σύστημα, καθώς επίσης και επειδή είναι αντιπροσωπευτικές για άλλα πολλά επερωτήματα που χρησιμοποιούνται σε Βάσεις Δεδομένων. Σημαντικό είναι να αναφέρουμε ότι τα δεδομένα που χρησιμοποιούμε ώς είσοδο για αυτές τις εφαρμογές, τα παράγουμε μέσω της εφαρμογής DBGEN (Database Generator). Η συγκεκριμένη εφαρμογή, μπορεί να δημιουργήσει τα αρχεία - πίνακες (tables) που χρειάζονται για την εκτέλεση ενός επερωτήματος από την σουίτα TPC-H. Επίσης, μέσω του DBGEN ο χρήστης έχει την ευκαιρία να δημιουργήσει αρχεία διαφορετικών μεγεθών, φτάνει να ορίσει μέσω συγκεκριμένων επιλογών το μέγεθος που θέλει.

- **TPC-H Q3 (Query-3)**

  Αυτή η εφαρμογή, αποτελείται από 3 λειτουργίες σάρωσης και 2 λειτουργίες σύνδεσης. Αυτές οι λειτουργίες, φαίνονται ξεκάθαρα στον

SQL κώδικα του σχήματος 4.2. Η μία λειτουργία σάρωσης, αφορά την επιλογή των πλειάδων του πίνακα LINEITEM για τις οποίες ισχύει η συνθήκη **l_shipdate > date.** Η άλλη λειτουργία σάρωσης, επιλέγει τις πλειάδες του πίνακα ORDERS που πληρούν την συνθήκη **o_orderdate < date**, ενώ η τρίτη λειτουργία σάρωσης, επιλέγει τις πλειάδες του πίνακα CUSTOMER για τις οποίες ισχύει η συνθήκη **c_mktsegment = '[SEGMENT]'**. Όσο αφορά τις λειτουργίες σύνδεσης, η μία απο αυτές γίνεται μεταξύ των πινάκων LINEITEM και ORDERS, ενώ η άλλη μεταξύ των πινάκων ORDERS και CUSTOMER.

Στην ουσία, η εφαρμογή αυτή ανακτά τις παραγγελίες που:

1. Δεν έχουν σταλεί πριν μία καθορισμένη ημερομηνία που δίνεται ώς είσοδος.
2. Η παραγγελία τους προηγείται μίας ημερομηνίας που δίνεται ώς είσοδος.

Για αυτή την εφαρμογή, φτιάξαμε δύο διαφορετικές υλοποιήσεις, εκ των οποίων η μία χρησιμοποιά την τεχνική του “nested-loop join” και η άλλη την τεχνική του “hash join”. Στην συνέχεια της διπλωματικής μου εργασίας, όταν αναφερόμαστε στο Query-3 θα εννοούμε την υλοποίηση με το “nested-loop join”, ενώ όταν αναφερόμαστε στο Query-3-hashing θα εννοούμε την υλοποίηση με “hash join”.

```
select
        l_orderkey ,
from
        customer , orders , lineitem
where
        c_mktsegment = '[SEGMENT]'
        and c_custkey = o_custkey
        and l_orderkey = o_rderkey
        and o_orderdate < date '[DATE]'
        and l_shipdate > date '[DATE]';
```

Σχήμα 4.2: *Ο SQL κώδικας του TPC-H Q3*

- **TPC-H Q12 (Query-12)**

  Αυτή η εφαρμογή, αποτελείται από 2 λειτουργίες σάρωσης και μια λειτουργία σύνδεσης. Η μία λειτουργία σάρωσης, επιλέγει τις πλειάδες του πίνακα LINEITEM για τις οποίες ισχύουν κάποιες συνθήκες που έχουν να κάνουν με την ημερομηνία παραλαβής μίας παραγγελίας, ενώ η άλλη λειτουργία επιλέγει τις παραγγελίες (πλειάδες) του πίνακα ORDERS που έχουν επείγουσα ή υψηλή προτεραιότητα (o_orderpriority = '1-URGENT' or o_orderpriority = '2-HIGH').

  Η σύνδεση εκτελείται μεταξύ των πινάκων LINEITEM και ORDERS. Η έξοδος που παράγει αυτό το επερώτημα, είναι το άθροισμα των παραγγελιών που πληρούν τις πιο πάνω συνθήκες. Στο πιο κάτω σχήμα (σχήμα 4.3) παρουσιάζουμε τον SQL κώδικα της συγκεκριμένης εφαρμογής.

  Για αυτή την εφαρμογή σε αυτή την διπλωματική εργασία, παρουσιάζουμε τα αποτελέσματα μόνο για την υλοποίηση με "nested-loop join" και όχι για την υλοποίηση με "hash join" όπως στο query-3. Αυτό επειδή αυτή η εφαρμογή, έχει μόνο μια λειτουργία σύνδεσης και συνεπώς ελάχιστο χρόνο **σειριακής** εκτέλεσης ακόμα και για τεράστια μεγέθη αρχείων.

```
select
        sum(case when
                o_orderpriority = '1-URGENT' or
                o_orderpriority = '2-HIGH'
                then 1 else 0 end)
         as high_line_count
from
        orders , lineitem
where
        o_rderkey = l_orderkey
        and l_shipmode in
        ('[SHIPMODE1]', '[SHIPMODE1]')
        and l_commitdate < l_receiptdate
        and l_shipdate < l_commitdate
        and l_receiptdate > date '[DATE]'
        and l_receiptdate > date '[DATE]'
        + interval '1' year;
```

Σχήμα 4.3: *Ο SQL κώδικας του TPC-H Q12*

Ο κώδικας των εφαρμογών που χρησιμοποιούμε, είναι γραμμένος στην γλώσσα προγραμματισμού C και για τον παραλληλισμό χρησιμοποιούμε την βιβλιοθήκη των PThreads. Για την δημιουργία των νημάτων κάνουμε χρήση της κλήσης pthread_create() και για τον συγχρονισμό μεταξύ των νημάτων χρησιμοποιούμε κλειδαριές (locks), τις οποίες κλειδώνουμε και ξεκλειδώνουμε με τις κλήσεις pthread_mutex_lock() και pthread_mutex_unlock() αντίστοιχα.

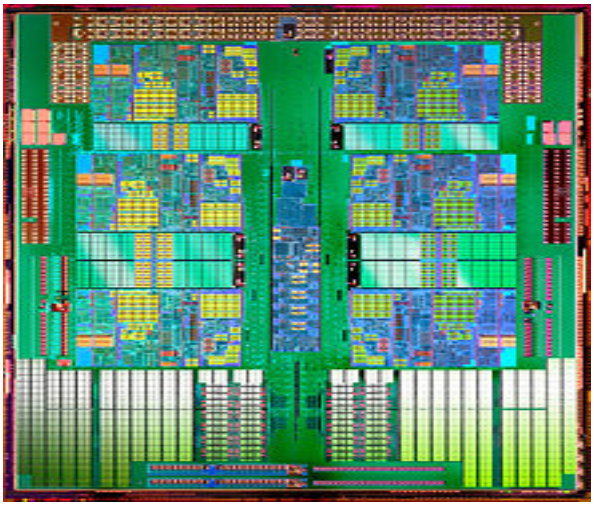
Το συνολικό μέγεθος όλων των αρχείων που δίνουμε σάν είσοδο στην κάθε εφαρμογή, φαίνεται στον πίνακα 4.2 (Total Size). Επίσης, εκτός από το συνολικό μέγεθος, παρουσιάζουμε στον ίδιο πίνακα και το μέγεθος κάθε ξεχωριστού αρχείου εισόδου. Στην εφαρμογή Query-3-Hashing, τα μεγέθη των αρχείων εισόδου είναι πολύ μεγάλα, λόγω του ότι με μικρά μεγέθη ο σειριακός χρόνος εκτέλεσης της εφαρμογής είναι ελάχιστος. Αυτό οφείλεται λόγω της τεχνικής του κατακερματισμού που χρησιμοποιείται στην συγκεκριμένη εφαρμογή, την οποία περιγράψαμε πιο πάνω σε προηγούμενο υποκεφάλαιο.
Τα δεδομένα αυτών των αρχείων, κατα την διάρκεια εκτέλεσης αποθηκεύονται σε στατικούς πίνακες στις εφαρμογές Query-12 και Query-3, ενώ στην εφαρμογή Query-3-Hashing σε δυναμικούς πίνακες και ακολούθως στον ανάλογο πίνακα κατακερματισμού.
Σε όλα μας τα πειράματα μετράμε τον χρόνο εκτέλεσης μόνο του παράλληλου μέρους κάθε εφαρμογής, αφού αυτό είναι που μας ενδιαφέρει. Σειριακές εργασίες όπως το διάβασμα αρχείων ή εκτυπώσεις αποτελεσμάτων δεν λαμβάνονται υπόψιν στον χρόνο εκτέλεσης.

Benchmark	**Lineitem Table Size**	**Orders Table Size**	**Customer Table Size**	**Total Size**
QUERY-12	70.80 MB	16.10 MB	-	86.90 MB
QUERY-3	70.80 MB	16.10 MB	2.30 MB	89.20 MB
QUERY-3-HASHING	18.30 GB	4.10 GB	0.57 GB	22.97 GB

Πίνακας 4.2: *Μεγέθη αρχείων εισόδου για κάθε εφαρμογή*

4.3 Περιγραφή Συστήματος Αξιολόγησης

Για την πειραματική αξιολόγηση των εφαρμογών μας, χρησιμοποιήσαμε την μηχανή cs6472 του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου. Η συγκεκριμένη μηχανή αποτελείται από δύο εξαπύρηνους AMD Opteron 2427 επεξεργαστές. Αυτό σημαίνει ότι η μηχανή αποτελείται απο συνολικά 12 πυρήνες. Κάθε ένας από αυτούς τους πυρήνες, έχει συχνότητα 2.194612 GHz και την δική του κρυφή μνήμη (cache) χωρητικότητας 512 KB (kilobyte). Η μνήμη RAM της συγκεκριμένης μηχανής έχει συνολικό μέγεθος 31 GB (gigabyte).



Σχήμα 4.3: *Εξαπύρηνος AMD Opteron Επεξεργαστής [24]*

Το λειτουργικό σύστημα που είναι εγκατεστημένο στην μηχανή cs6472, είναι τα 64-bit Linux 3.2.0-38-generic. Ο μεταγλωττιστής που χρησιμοποιήθηκε για την μεταγλώττιση των εφαρμογών στην γλώσσα C, είναι ο gcc 4.6.3.

4.4 Περιγραφή Τρόπου Αξιολόγησης

Για την αξιολόγηση των υλοποιήσεών μας που περιγράφουμε στο επόμενο κεφάλαιο, χρησιμοποιούμε την μετρική της επιτάχυνσης (speedup). Στην Παράλληλη Επεξεργασία, η επιτάχυνση αντιπροσωπεύει το πόσες φορές είναι

γρηγορότερος ένας παράλληλος αλγόριθμος απο τον αντίστοιχο σειριακό αλγόριθμο. Η φόρμουλα για τον υπολογισμό της επιτάχυνσης, είναι η εξής:

$$SPEEDUP_n = T_1 / T_n$$

Όπου:

n : Αριθμός των νημάτων στον παράλληλο αλγόριθμο.

T_1 : Χρόνος εκτέλεσης σειριακού αλγορίθμου.

T_n : Χρόνος εκτέλεσης παράλληλου αλγορίθμου με n-νήματα.

$SPEEDUP_n$: Επιτάχυνση που επιτυγχάνεται με χρήση n-νημάτων.

Η καλύτερη δυνατή επιτάχυνση (ιδεατή) που μπορεί να επιτευχθεί μέσω του παραλληλισμού μιας εφαρμογής, είναι ίση με τον αριθμό των νημάτων που χρησιμοποιούνται για την εκτέλεση αυτής της εφαρμογής. Φυσικά, κάποιες φορές ενδέχεται η επιτάχυνση μιας παράλληλης εφαρμογής να είναι μεγαλύτερη απο ιδεατή. Σε αυτή την περίπτωση, η επιτάχυνση ονομάζεται “super linear speedup”. Ο κυριότερος λόγος για τον οποίο κάποιες φορές επιτυγχάνεται super linear speedup, είναι η σημαντική μείωση των “cache misses” (σε σχέση με την σειριακή εκτέλεση), αφού τα δεδομένα διαμοιράζονται μεταξύ των νημάτων, κάθε νήμα τρέχει σε ένα πυρήνα και έχει την δική του κρυφή μνήμη στην οποία μεταφέρει πιο λίγο αριθμό δεδομένων απο ότι τα δεδομένα που μεταφέρονται στην κρυφή μνήμη του κύριου νήματος στην σειριακή εκτέλεση.

Κεφάλαιο 5

Βάση και Ιδέα Υλοποίησης

5.1 Ιδέα και Βάση Υλοποίησης με Παραλληλισμό σε Επίπεδο Δεδομένων και Στατικό Καταμερισμό Εργασίας

Η βασική ιδέα η οποία διέπει αυτή την προσέγγιση, είναι όλα τα νήματα που δημιουργούνται για την εκτέλεση της κάθε εφαρμογής, να εκτελούν τους ίδιους υπολογισμούς (ίδιο κομμάτι κώδικα), σε διαφορετικό όμως σύνολο απο δεδομένα. Αυτό που κάνουμε λοιπόν σε αυτή την υλοποίηση, είναι πρώτα να μοιράσουμε στατικά τα δεδομένα μεταξύ των 12 νημάτων. Ακολούθως, κάθε νήμα εκτελεί όλες τις πράξεις και τους υπολογισμούς πάνω στο σύνολο δεδομένων που του έχει ανατεθεί και υπολογίζει το δικό του τοπικό αποτέλεσμα. Όταν υπολογίσει το τοπικό αποτέλεσμά του, τότε γράφει το αποτέλεσμα σε ένα κοινόχρηστο πίνακα. Όταν όλα τα νήματα παράγουν το αποτέλεσμά τους, τότε το κύριο νήμα (main thread) προσθέτει αυτά τα αποτελέσματα για να υπολογίσει και να παρουσιάσει το τελικό αποτέλεσμα.
Στο σχήμα 5.1, παρουσιάζουμε τον κώδικα του Query-12 τον οποίο εκτελεί κάθε νήμα προκειμένου να υπολογίσει το δικό του τοπικό αποτέλεσμα. Η ίδια ιδέα και βάση υλοποίησης εφαρμόζεται στον κώδικα του Query-3 και του Query-3-Hashing. Η μεταβλητή LINEITEM αντιπροσωπεύει το μέγεθος του πίνακα lineitem, του οποίου τα δεδομένα μοιράζονται μεταξύ των νημάτων.

```
chunk = LINEITEM / NUMBER_OF_THREADS;
start = thread->id * chunk;

if(LINEITEM % NUMBER_OF_THREADS == 0)
    end = start + chunk;
else
{
    if(thread->id == (NUMBER_OF_THREADS - 1))
        end = start + chunk + (LINEITEM % NUMBER_OF_THREADS);
    else
        end = start + chunk;
}

for(i = start; i < end; i++)
{
    if((lineitem[i][4] >= 1994) && (lineitem[i][4] < 1995))
    {
        for(j = 0; j < ORDERS; j++)
        {
            if((lineitem[i][0] == orders[j][0]))
            {
                sum++;
            }
        }
    }
}
```

Σχήμα 5.1: *Ψευδοκώδικας χρήσης παραλληλισμού σε επίπεδο δεδομένων με στατικό καταμερισμό εργασίας στο Query-12.*

5.2 Ιδέα και Βάση Υλοποίησης με Παραλληλισμό σε Επίπεδο Δεδομένων και Δυναμικό Καταμερισμό Εργασίας

Σε αυτή την προσέγγιση, όπως και στην προηγούμενη προσέγγιση της οποίας ο καταμερισμός εργασίας ήταν στατικός, όλα τα νήματα εκτελούν την ίδια εργασία (όλους τους υπολογισμούς) πάνω σε διαφορετικό όμως σύνολο δεδομένων. Προχωρήσαμε στην υλοποίηση αυτής της προσέγγισης, λόγω του ότι ο στατικός καταμερισμός εργασίας αντιμετωπίζει ένα σημαντικό πρόβλημα. Το πρόβλημα αυτό, είναι το εξής. Όταν ένα ή περισσότερα νήματα τελειώσουν την εργασία τους πιο γρήγορα απο κάποια άλλα νήματα, παραμένουν χωρίς

εργασία. Τα άλλα όμως νήματα, συνεχίζουν να δουλεύουν, αφού ακόμα δεν έχουν τελειώσει με την εργασία που τους έχει ανατεθεί. Στην περίπτωση των εφαρμογών που χρησιμοποιήσαμε, αυτό συμβαίνει για ένα σημαντικό λόγο. Κάθε νήμα εκτελεί ένα βρόγχο επανάληψης στα δεδομένα ενός πίνακα τα οποία του αναλογούν. Για κάθε στοιχείο του πίνακα που διαβάζει, εάν ισχύει κάποια συγκεκριμένη συνθήκη τότε το νήμα εκτελεί ένα δεύτερο βρόγχο επανάληψης σε όλα τα στοιχεία ενός άλλου πίνακα. Αυτό σημαίνει ότι, εάν ισχύει η συνθήκη αυτή πολύ λιγότερες φορές σε ένα νήμα (έστω νήμα_X) παρά σε ένα άλλο νήμα (έστω νήμα_Y), τότε το νήμα_X θα εκτελέσει πιο γρήγορα την εργασία του, αφού θα εκτελέσει τον $2^{ο}$ βρόγχο επανάληψης λιγότερες φορές από το νήμα_Y. Συνεπώς, η προσέγγιση αυτή υστερεί στο ότι δεν εκμεταλλεύεται στο έπακρο τον παραλληλισμό όταν κάποια νήματα τελειώνουν την εργασία τους πιο γρήγορα απο τα υπόλοιπα νήματα.

Η διαφορά με την προηγούμενη προσέγγιση, είναι ότι σε αυτή την προσέγγιση χωρίζουμε τα δεδομένα του πίνακα lineitem σε ένα μεγάλο αριθμό ομάδων απο ίσο αριθμό δεδομένων (μία ομάδα έχει ίσο αριθμό δεδομένων με κάθε άλλη ομάδα). Κάθε νήμα, επιλέγει μία ομάδα δεδομένων και εκτελεί όλους τους υπολογισμούς χρησιμοποιώντας αυτά τα δεδομένα. Όταν τελειώσει, ελέγχει εάν υπάρχει ομάδα δεδομένων που ακόμα δεν έτυχε επεξεργασίας απο κάποιο άλλο νήμα και εκτελεί πάλι όλους τους υπολογισμούς χρησιμοποιώντας τη νέα ομάδα δεδομένων. Αυτό επαναλαμβάνεται έως ότου δεν υπάρχει άλλη ομάδα δεδομένων που να μην έχει τύχει επεξεργασίας.
Με αυτό τον τρόπο, επιτυγχάνεται πιο δίκαιος καταμερισμός εργασίας μεταξύ των νημάτων, αφού τα πιο γρήγορα απο αυτά θα εκτελέσουν περισσότερη εργασία απο τα πιο αργά νήματα (ο λόγος για τον οποίο κάποιο νήμα μπορεί να είναι πιο γρήγορο απο ένα άλλο έχει εξηγηθεί πιο πάνω). Αυτό έχει ώς αποτέλεσμα την βελτίωση της επίδοσης των εφαρμογών και συνεπώς την αύξηση της επιτάχυνσης. Για να κατανοήσουμε καλύτερα το πλεονέκτημα και την σημαντικότητα αυτής της προσέγγισης, ας παρατηρήσουμε τον κώδικα του Query-12 στο σχήμα 5.1, όπου έχουμε στατικό καταμερισμό εργασίας. Αν ο αριθμός των φορών που επιστρέφει TRUE η συνθήκη που βρίσκεται πρίν την εκτέλεση του εσωτερικού βρόγχου επανάληψης είναι πολύ μεγαλύτερος σε

κάποια νήματα παρά σε άλλα, τότε λογικά κάποια νήματα θα τελειώσουν την εργασία τους πρίν από τα άλλα νήματα. Αυτό, έχει ώς συνέπεια την μή επίτευξη ιδεατής επιτάχυνσης. Ο δυναμικός καταμερισμός εργασίας, καταπολεμά αυτό το μειονέκτημα, αφού αν ένα νήμα για παράδειγμα τελειώσει με μια ομάδα δεδομένων πιο γρήγορα απο ότι ένα άλλο νήμα, τότε θα εκτελέσει την εργασία του ξανά σε άλλη ομάδα δεδομένων κ.ο.κ, με αποτέλεσμα να επιτυγχάνει καλύτερη εκμετάλλευση του παραλληλισμού που παρέχεται απο το σύστημα.

5.3 Ιδέα και Βάση Υλοποίησης με Χρήση Στατικής Διασωλήνωσης

Για την υλοποίηση στατικής διασωλήνωσης, χρησιμοποιήσαμε το μοντέλο παραγωγού-καταναλωτή. Η βασική ιδέα είναι να ομαδοποιήσουμε τα νήματα σε δύο ομάδες: τους παραγωγούς και τους καταναλωτές. Οι παραγωγοί, εκτελούν μία εργασία και γράφουν την έξοδό τους σε ένα κοινόχρηστο πίνακα. Απο την άλλη, οι καταναλωτές διαβάζουν από τον πίνακα όταν εισαχθεί σε αυτόν κάποιο στοιχείο από τους παραγωγούς και χρησιμοποιούν αυτό το στοιχείο για τον υπολογισμό του τελικού αποτελέσματος της εφαρμογής.

Στον δικό μας στατικό αλγόριθμο, δημιουργούμε ζεύγη από νήματα-παραγωγούς και νήματα-καταναλωτές. Αντί να έχουμε ένα κοινό πίνακα μεταξύ όλων των ζευγών, ορίζουμε ένα πίνακα για κάθε ξεχωριστό ζεύγος από παραγωγούς - καταναλωτές. Τα ζεύγη αυτά, μοιράζονται τα δεδομένα που χρησιμοποιά η εφαρμογή για τον υπολογισμό του αποτελέσματος στατικά. Δηλαδή, έχουμε παραλληλισμό δεδομένων μεταξύ των ζευγών που δημιουργούνται, αφού κάθε ζεύγος εκτελεί την ίδια εργασία αλλά σε διαφορετικά δεδομένα. Η διασωλήνωση εφαρμόζεται εντός του κάθε ζεύγους.

Για να καταλάβουμε καλύτερα, έστω ότι έχουμε 6 παραγωγούς και 6 καταναλωτές. Αυτό σημαίνει ότι έχουμε συνολικά 6 ζεύγη, όπου κάθε ζεύγος αποτελείται απο 1 παραγωγό και 1 καταναλωτή. Κάθε τέτοιο ζεύγος έχει τον δικό του πίνακα. Στην περίπτωση που έχουμε 8 παραγωγούς και 4 καταναλωτές, τότε έχουμε συνολικά 4 ζεύγη, όπου κάθε ζεύγος αποτελείται από 2 παραγωγούς και 1 καταναλωτή. Αυτά τα αναφέρουμε, ώστε να μπορεί ο αναγνώστης να αντιληφθεί όσο πιο καλά γίνεται την ιδέα που διέπει τον στατικό αλγόριθμο που χρησιμοποιούμε.

```
producer(producerThread)
{
     myID := producerThread->id;

     for(i := start; i < end; i++)
     {
          item := produce_item();

          write_position := i;
          insert_into_buffer(buffer[myID], write_position, item);

          producerSema[myID]++;
     }
}
```

Σχήμα 5.2(α): *Ψευδοκώδικας που εκτελεί ένα νήμα που έχει ρόλο παραγωγού*

Στο σχήμα 5.2(α), παρουσιάζουμε τον ψευδοκώδικα ενός παραγωγού για την στατική διασωλήνωση, ενώ στο σχήμα 5.2(β) τον ψευδοκώδικα ενός καταναλωτή. Ο συγκεκριμένος ψευδοκώδικας, δουλεύει για ίσο αριθμό παραγωγών-καταναλωτών, αλλά δουλεύει και για όλες τις άλλες περιπτώσεις (π.χ περισσότεροι παραγωγοί παρά καταναλωτές) με μόνο ελάχιστες και απλές μετατροπές στον κώδικα.

Ένα νήμα-παραγωγός, όπως βλέπουμε στον ψευδοκώδικα του σχήματος 5.2(α) έχει την δική του ταυτότητα (producerThread->id). Το ίδιο και ένας καταναλωτής (consumerThread->id) όπως παρατηρούμε από τον ψευδοκώδικα του σχήματος 5.2(β). Ένας παραγωγός και ένας καταναλωτής οι οποίοι ανήκουν στο ίδιο ζεύγος, έχουν την ίδια ταυτότητα και μοιράζονται τον ίδιο κοινόχρηστο πίνακα (buffer[ID]).

```
consumer(consumerThread)
{
     myID := consumerThread->id;

     for(i := start; i < end; i++)
     {
          /* busy wait */
          while(consumerSema[myID] == producerSema[myID]);

          read_position := i;
          item := get_from_buffer(buffer[myID], read_position);
          consume_item(item);

          consumerSema[myID]++;
     }
}
```

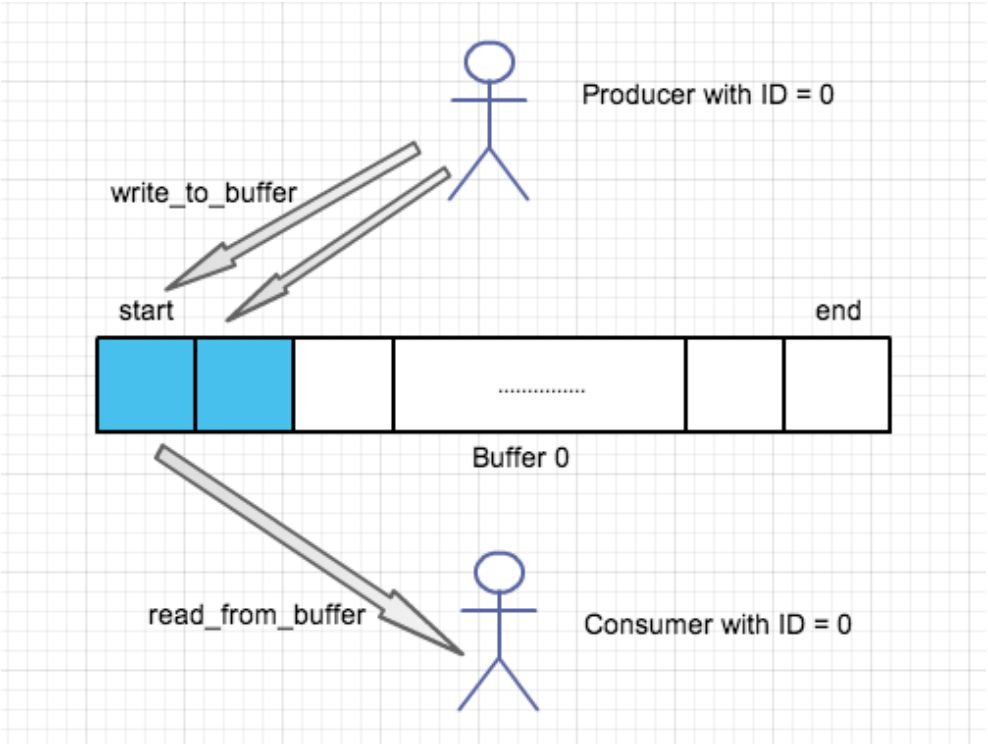
Σχήμα 5.2(β): *Ψευδοκώδικας που εκτελεί ένα νήμα που έχει ρόλο καταναλωτή*

Όπως παρατηρούμε στον ψευδοκώδικα ενός παραγωγού, ο παραγωγός εκτελεί κάποια εργασία και παράγει κάποιο αντικείμενο (item), το οποίο τοποθετεί στην συνέχεια στον πίνακα που του αναλογεί. Η θέση στην οποία τοποθετείται αυτό το αντικείμενο, αντιπροσωπεύεται από την μεταβλητή write_position η οποία περνιέται σαν παράμετρος στην συνάρτηση insert_into_buffer, ώστε να εισαχθεί το αντικείμενο στην συγκεκριμένη θέση. Αφού τοποθετήσει το αντικείμενο που έχει παράξει στον πίνακα, αυξάνει τον σημαφόρο του κατά 1.

Απο την άλλη, ο καταναλωτής, όπως βλέπουμε στον ψευδοκώδικά του, προτού διαβάσει από μία θέση του πίνακα, ελέγχει εάν ο σημαφόρος του ισούται με τον σημαφόρο του παραγωγού στον οποίο αναλογεί. Εάν οι δύο σημαφόροι είναι ίσοι, τότε σημαίνει ότι ο παραγωγός δέν έχει ακόμα τοποθετήσει κάποιο αντικείμενο στην συγκεκριμένη θέση και έτσι ο καταναλωτής περιμένει μέχρι να τοποθετηθεί κάποιο αντικείμενο στην θέση αυτή. Ο συγχρονισμός αυτός επιτυγχάνεται με **busy wait**, μέσω της κλήσης ***while(consumerSema[myID] == producerSema[myID])***. Όταν τοποθετηθεί κάποιο αντικείμενο στην θέση αυτή από τον παραγωγό, τότε ο καταναλωτής διαβάζει από τον πίνακα, καταναλώνει το συγκεκριμένο αντικείμενο και ακολούθως αυξάνει τον σημαφόρο του κατά 1.

Η διαδικασία αυτή επαναλαμβάνεται από ένα ζεύγος παραγωγού-καταναλωτή, έως ότου τελειώσουν με το μέρος της εργασίας που τους έχει ανατεθεί.

Στο σχήμα 5.3 βλέπουμε σχηματικά τον τρόπο με τον οποίο δουλεύει η στατική διασωλήνωση για ένα ζεύγος παραγωγού-καταναλωτή, συγκεκριμένα του παραγωγού και του καταναλωτή με ταυτότητα ίση με μηδέν.



Σχήμα 5.3: *Ζεύγος παραγωγού-καταναλωτή*

Το ερώτημα που μας απασχόλησε σε αυτή την υλοποίηση, αφορά τον αριθμό των παραγωγών και καταναλωτών που ορίζουμε σε κάθε ζεύγος. Για αυτό τον λόγο δοκιμάσαμε διάφορους συνδυασμούς, τους οποίους παρουσιάζουμε στο κεφάλαιο 6, για να βρούμε τον καλύτερο δυνατό συνδυασμό. Απο τα αποτελέσματα που προέκυψαν, καταλήξαμε στο συμπέρασμα ότι δεν υπάρχει μια “μαγική συνταγή” για τον αριθμό των παραγωγών και των καταναλωτών που πρέπει να ορίζεται. Ο αριθμός των παραγωγών και καταναλωτών που πρέπει να ορίζεται, εξαρτάται απο την εφαρμογή και τον συνολικό φόρτο εργασίας που έχει κάθε ομάδα (παραγωγών - καταναλωτών). Δηλαδή, σε μια εφαρμογή μπορεί να συμφέρει να χρησιμοποιείται μεγαλύτερος αριθμός παραγωγών απο ότι καταναλωτών, ενώ σε άλλη εφαρμογή μεγαλύτερος αριθμός καταναλωτών παρά παραγωγών, ανάλογα του φόρτου εργασίας της κάθε ομάδας. Για να βρεθεί ο καλύτερος συνδυασμός παραγωγών - καταναλωτών, πρέπει να γίνει μελέτη για το ποια ομάδα έχει περισσότερη δουλειά απο την άλλη και έτσι να δοθούν περισσότερα νήματα στην μία εκ των δύο ομάδων εάν χρειάζεται.

Η ανάγκη για αποφυγή της μελέτης για το ποια εκ των δύο ομάδων είναι πιο αργή (λόγω μεγαλύτερου φόρτου εργασίας) με σκοπό να της δώσουμε

περισσότερα νήματα, αλλά και η ανάγκη για μεγαλύτερη επίδοση μας οδήγησε στην δημιουργία ενός αλγόριθμου που κάνει χρήση δυναμικής διασωλήνωσης, τον οποίο παρουσιάζουμε πιο κάτω.

5.4 Ιδέα και Βάση Υλοποίησης με Χρήση Δυναμικής Διασωλήνωσης

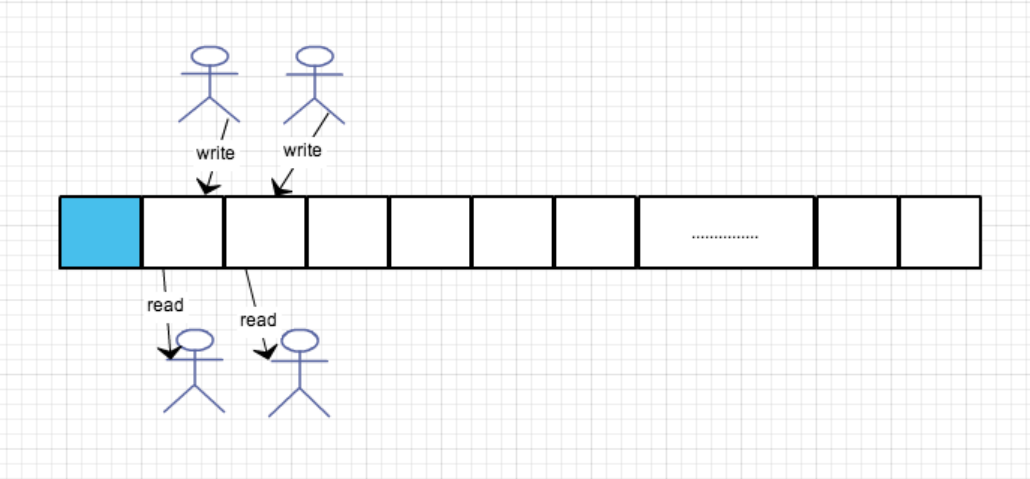
Προκειμένου να πετύχουμε υψηλή επίδοση στις εφαρμογές μας με την χρήση της υλοποίησης με στατική διασωλήνωση που περιγράψαμε στο προηγούμενο υποκεφάλαιο, πρέπει να ορίσουμε κατάλληλα τον αριθμό των παραγωγών και τον αριθμό των καταναλωτών ώστε να μήν είναι κάποια από τις δύο ομάδες γρηγορότερη ή πιο αργή απο την άλλη. Αν κάποια ομάδα (παραγωγοί ή καταναλωτές) είναι πιο αργή απο ότι η άλλη ομάδα, πρέπει να της δίνουμε περισσότερα νήματα απο ότι στην άλλη ομάδα, ώστε να έχουμε καλύτερη εκμετάλλευση του παραλληλισμού και συνεπώς μεγαλύτερη και καλύτερη επίδοση. Ο κυριότερος λόγος για τον οποίο μία ομάδα μπορεί να είναι πιο αργή απο την άλλη ομάδα, είναι ο μεγαλύτερος αριθμός πράξεων και ο φόρτος εργασίας που εκτελεί.

Σε αυτό τον αλγόριθμο, έχουμε ένα μόνο κοινό πίνακα (buffer) στον οποίο πρόσβαση έχουν όλοι οι παραγωγοί και όλοι οι καταναλωτές. Επίσης, δεν έχουμε ζεύγη απο παραγωγούς-καταναλωτές όπως είχαμε στον στατικό αλγόριθμο που περιγράψαμε πιο πάνω. Ο δυναμικός αλγόριθμος, αρχικά, ξεκινάει με ίσο αριθμό παραγωγών και καταναλωτών (στην δική μας περίπτωση με 6 παραγωγούς και 6 καταναλωτές). Κατα την διάρκεια που εκτελείται η εφαρμογή, ένα νήμα που έχει ρόλο καταναλωτή, αυξάνει κάποιο τοπικό μετρητή κάθε φορά που παραμένει χωρίς δουλειά λόγω του ότι δεν έχει τοποθετηθεί ακόμα κάποιο αποτέλεσμα από τους παραγωγούς. Αν ο μετρητής αυτός ξεπεράσει ένα όριο (threshold), τότε σημαίνει ότι η ομάδα των καταναλωτών είναι πιο γρήγορη απο αυτή των παραγωγών και έτσι ο συγκεκριμένος καταναλωτής γίνεται παραγωγός. Με αυτό τον τρόπο, επιτυγχάνουμε την αύξηση του αριθμού των παραγωγών στην περίπτωση που είναι πιο αργοί από τους καταναλωτές. Ο αλγόριθμος μας διασφαλίζει πάντα την ύπαρξη ενός τουλάχιστον καταναλωτή καθ'όλη την διάρκεια εκτέλεσης μιας εφαρμογής, για να μπορεί να υφίσταται η διασωλήνωση καθ'όλη την διάρκεια της εκτέλεσης.

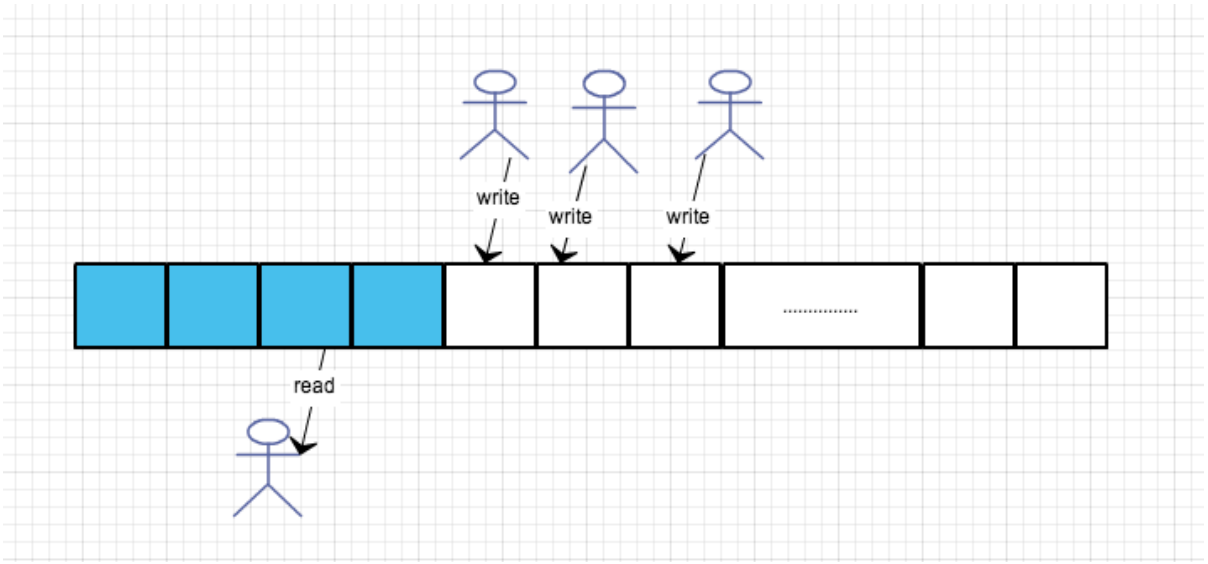
Απο την άλλη, όταν ένας παραγωγός τελειώνει την συνολική του εργασία, ελέγχει εάν οι καταναλωτές δέν έχουν τελειώσει με την εργασία τους. Εάν ισχύει αυτό, τότε γίνεται και αυτός καταναλωτής. Αυτό σημαίνει ότι εκμεταλλευόμαστε στο έπακρο τον παραλληλισμό που μας παρέχεται από το σύστημα, αφού κανένα νήμα δέν τερματίζει την εκτέλεσή του εάν ακόμα υπάρχουν νήματα που εκτελούν κάποια εργασία.

Στα σχήματα 5.4(α), 5.4(β) και 5.4(γ) μπορούμε να δούμε σχηματικά ένα σενάριο εκτέλεσης του αλγορίθμου μας. Στο σχήμα 5.4(α) ο αλγόριθμος ακόμα είναι σε αρχικό στάδιο με ίσο αριθμό παραγωγών και καταναλωτών (2 παραγωγοί – 2 καταναλωτές). Όπως παρατηρούμε από το ίδιο σχήμα, οι δύο καταναλωτές προσπαθούν να διαβάσουν από θέσεις του πίνακα στις οποίες δέν έχει εισαχθεί κάποιο αποτέλεσμα από τους παραγωγούς ακόμα. Αυτό σημαίνει ότι οι δύο καταναλωτές είναι πιο γρήγοροι από τους δύο παραγωγούς και αναγκαστικά σε κάποια φάση ο ένας απο αυτούς θα γίνει παραγωγός.

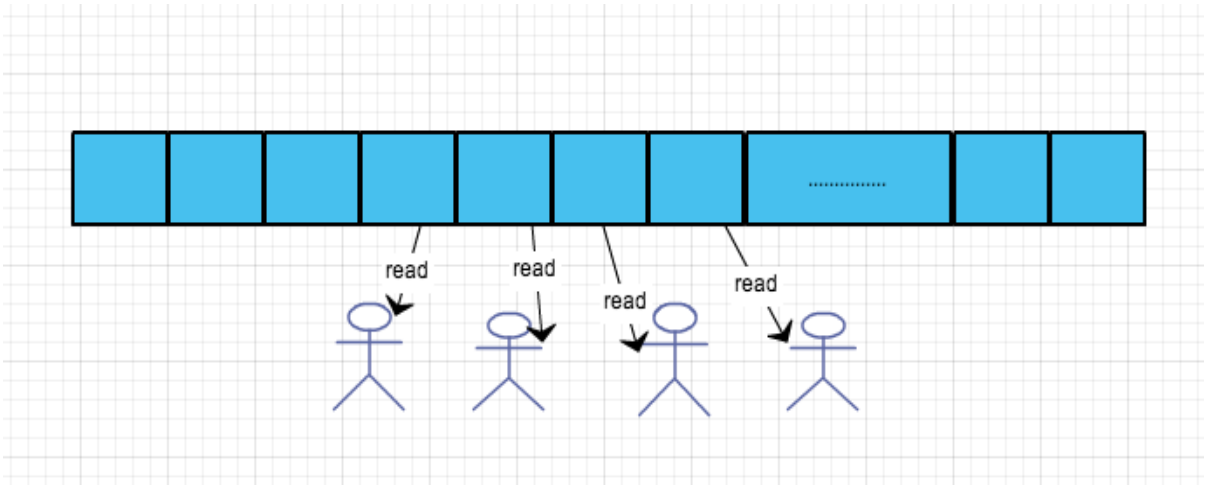
Αυτό φαίνεται ξεκάθαρα στην εικόνα 5.4(β), όπου αυξάνεται ο αριθμός των παραγωγών και μειώνεται αυτός των καταναλωτών. Αφού αυξήθηκε ο αριθμός των παραγωγών και ώς αποτέλεσμα έχουμε στο σενάριό μας το να τερματίσουν πιο γρήγορα από τον καταναλωτή, οι συγκεκριμένοι παραγωγοί θα γίνουν και αυτοί καταναλωτές. Αυτό φαίνεται στο σχήμα 5.4(γ).



Σχήμα 5.4(α): *Αρχικό στάδιο αλγορίθμου όπου έχουμε ίσο αριθμό παραγωγών και καταναλωτών.*



Σχήμα 5.4(β): *Αύξηση του αριθμού των παραγωγών κατα την διάρκεια εκτέλεσης του Αλγορίθμου.*



Σχήμα 5.4(γ): *Οι παραγωγοί αναλαμβάνουν ρόλο καταναλωτή αφού έχουν τελειώσει την εργασία τους.*

Κεφάλαιο 6

Πειραματική Αξιολόγηση

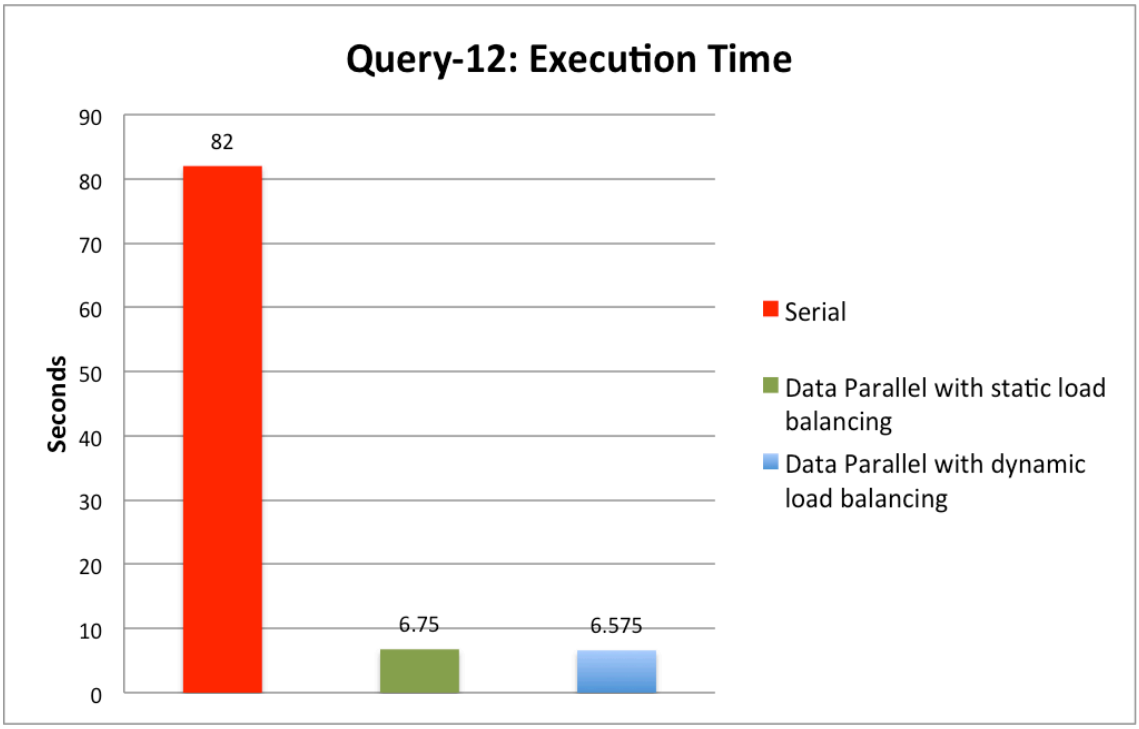
6.1 Παρουσίαση Αποτελεσμάτων για την Χρήση Παραλληλισμού σε Επίπεδο Δεδομένων

Όπως αναφέραμε στο υποκεφάλαιο “Στόχος και Αναμενόμενα Αποτελέσματα” του πρώτου κεφαλαίου, ο κυριότερος μας στόχος ήταν η επίτευξη ιδεατής επιτάχυνσης για τις εφαρμογές που θα χρησιμοποιούσαμε, χωρίς το τελικό αποτέλεσμα-έξοδος των εφαρμογών να είναι λανθασμένο. Πιο κάτω, παρουσιάζουμε τα αποτελέσματα που αφορούν τον χρόνο εκτέλεσης και την επιτάχυνση που επιτεύγχθηκε με την χρήση του παραλληλισμού σε επίπεδο δεδομένων στις εφαρμογές μας. Όπως προαναφέραμε στο κεφάλαιο “Πειραματική Διάταξη”, χρησιμοποιούμε 12 νήματα για όλα μας τα πειράματα.

6.1.1 Query-12

Στο σχήμα 6.1(α) παρουσιάζεται η γραφική παράσταση με τους χρόνους εκτέλεσης του σειριακού Query-12, του Query-12 με παραλληλισμό σε επίπεδο

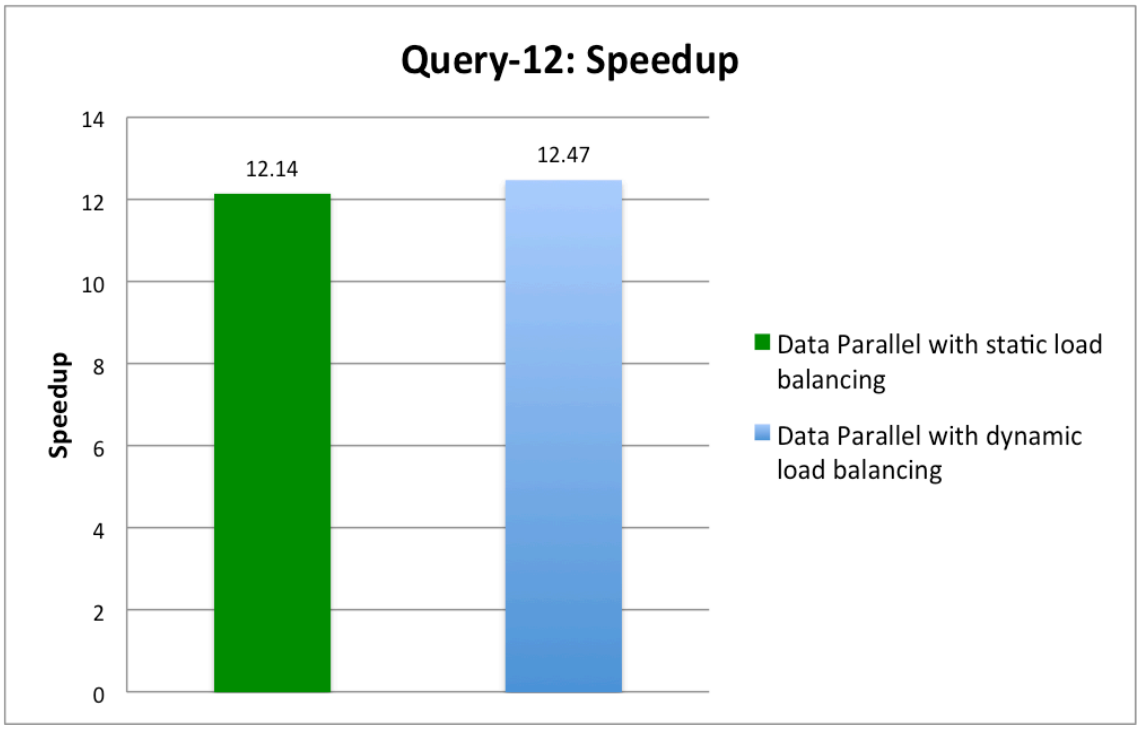
δεδομένων και στατικό καταμερισμό εργασίας και του Query-12 με παραλληλισμό σε επίπεδο δεδομένων και δυναμικό καταμερισμό εργασίας.



Σχήμα 6.1(α): *Χρόνοι εκτέλεσης για Query-12 με παραλληλισμό δεδομένων*

Όπως παρατηρούμε, ο χρόνος εκτέλεσης και για τις δύο προσεγγίσεις όπου χρησιμοποιούμε παραλληλισμό σε επίπεδο δεδομένων είναι πολύ μικρός σε σχέση με τον χρόνο εκτέλεσης του σειριακού Query-12. Επίσης, παρατηρούμε ότι ο χρόνος εκτέλεσης των δύο προσεγγίσεων που χρησιμοποιούν παραλληλισμό δεδομένων είναι σχεδόν ο ίδιος, με ελάχιστα μικρότερο τον χρόνο της προσέγγισης που χρησιμοποιά δυναμικό καταμερισμό εργασίας.

Στο σχήμα 6.1(β), παρουσιάζουμε την γραφική παράσταση που δείχνει την επιτάχυνση των δύο προσεγγίσεων με παραλληλισμό δεδομένων, σε σχέση με την σειριακή εκτέλεση.



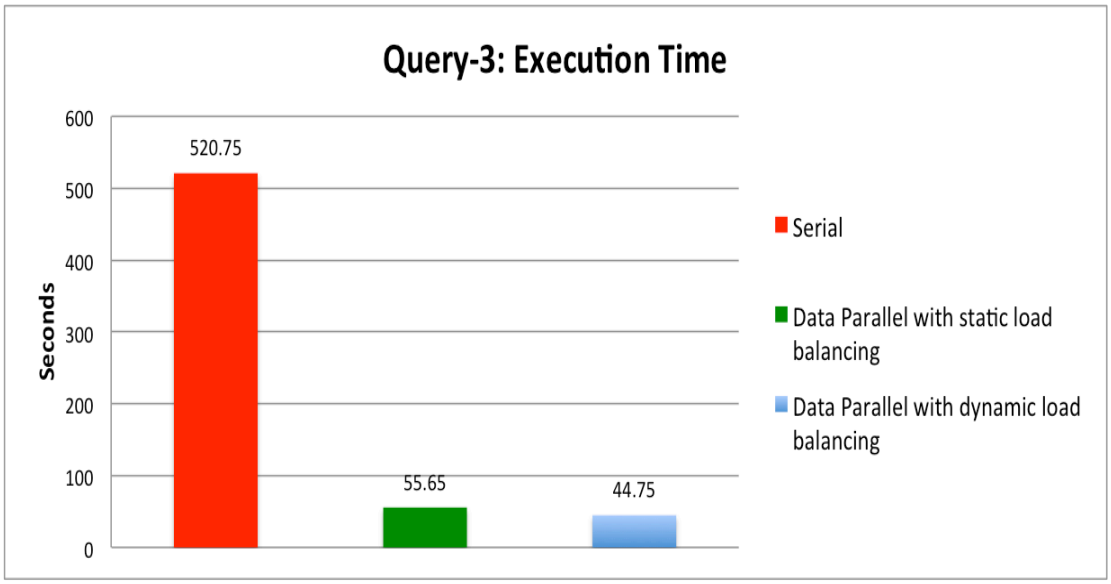
Σχήμα 6.1(β): *Επιτάχυνση για Query-12 με παραλληλισμό δεδομένων*

Όπως παρατηρούμε, η επιτάχυνση και στις δύο προσεγγίσεις, τόσο με στατικό τόσο και με δυναμικό καταμερισμό εργασίας είναι ελάχιστα πιο ψηλή απο ότι η ιδεατή επιτάχυνση. Αυτό συμβαίνει για τον εξής λόγο. Όταν η εκτέλεση είναι σειριακή, το κύριο νήμα (main thread) χρησιμοποιά όλα τα δεδομένα για τον υπολογισμό του αποτελέσματος, ενώ όταν έχουμε παραλληλισμό δεδομένων, τα δεδομένα μοιράζονται μεταξύ των 12 νημάτων που εκτελούν την εφαρμογή. Συνεπώς, ένα νήμα στην παράλληλη εκτέλεση έχει λιγότερα “cache misses” απο τα “cache misses” του κύριου νήματος στην σειριακή εκτέλεση (αφού χρησιμοποιά λιγότερα δεδομένα). Πιο συγκεκριμένα, μετά απο μετρήσεις των “cache misses” που έγιναν, ένα νήμα στην παράλληλη εκτέλεση έχει τα μισά “cache misses” απο αυτά που έχει το κύριο νήμα στην σειριακή εκτέλεση. Αυτός είναι και ο λόγος που έχουμε επιτάχυνση μεγαλύτερη απο την ιδεατή.

6.1.2 Query-3

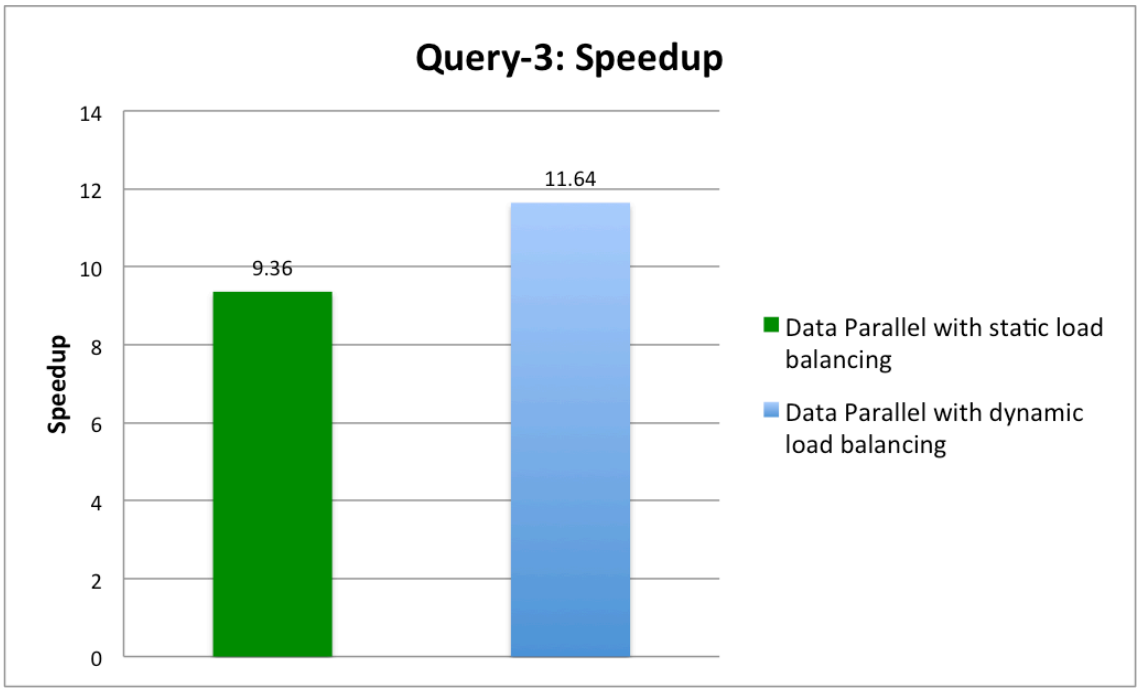
Στο σχήμα 6.2(α), παρουσιάζουμε την γραφική παράσταση με τον χρόνο εκτέλεσης του σειριακού Query-3, τον χρόνο εκτέλεσης του Query-3 με παραλληλισμό δεδομένων και στατικό καταμερισμό εργασίας και τον χρόνο

εκτέλεσης του Query-3 με παραλληλισμό δεδομένων και δυναμικό καταμερισμό εργασίας. Στο σχήμα 6.2(β), παρουσιάζουμε την επιτάχυνση που επιτυγχάνουμε με την χρήση του παραλληλισμού δεδομένων στο Query-3. Αυτό που παρατηρούμε από την γραφική παράσταση του σχήματος 6.2(α), είναι ότι ο χρόνος εκτέλεσης του Query-3 είναι πολύ μικρός και για τις δύο προσεγγίσεις όπου χρησιμοποιείται ο παραλληλισμός δεδομένων σε σχέση με τον χρόνο εκτέλεσης του σειριακού Query-3. Επίσης, όπως βλέπουμε, ο χρόνος εκτέλεσης με την χρήση δυναμικού καταμερισμού εργασίας είναι κατα 10 περίπου δευτερόλεπτα μικρότερος από τον χρόνο εκτέλεσης της προσέγγισης που χρησιμοποιεί στατικό καταμερισμό εργασίας. Αυτό αποδεικνύει την υπεροχή του δυναμικού έναντι του στατικού καταμερισμού εργασίας.



Σχήμα 6.2(α): *Χρόνοι εκτέλεσης για Query-3 με παραλληλισμό δεδομένων*

Η υπεροχή αυτή φαίνεται καλύτερα στην γραφική παράσταση του σχήματος 6.2(β). Όπως παρατηρούμε σε αυτή την γραφική παράσταση, η επιτάχυνση που επιτυγχάνεται με 12 νήματα και στατικό καταμερισμό εργασίας είναι 9,36. Η επιτάχυνση αυτή είναι ψηλή, αλλά όχι ιδεατή (αφού η ιδεατή ισούται με τον αριθμό των νημάτων που χρησιμοποιούνται κατά την εκτέλεση μιας εφαρμογής). Αυτό συμβαίνει για τον λόγο ότι κάποια νήματα τελειώνουν την εργασία τους πιο γρήγορα απο ότι άλλα νήματα, με αποτέλεσμα να μήν έχουμε πλήρη εκμετάλλευση του παραλληλισμού.

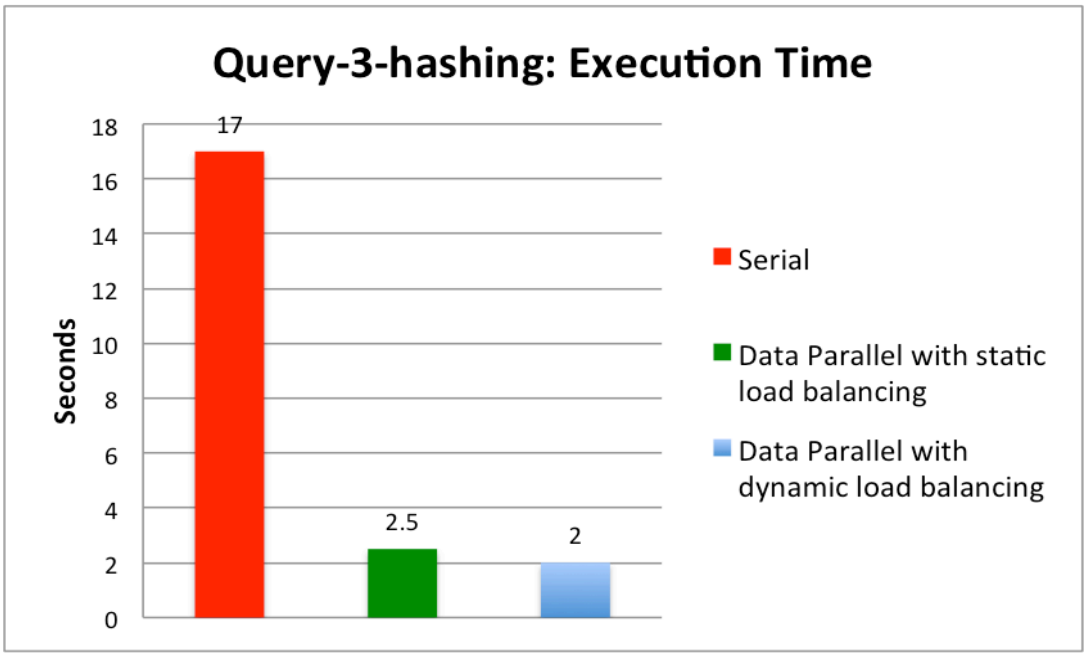


Σχήμα 6.2(β): *Επιτάχυνση για Query-3 με παραλληλισμό δεδομένων*

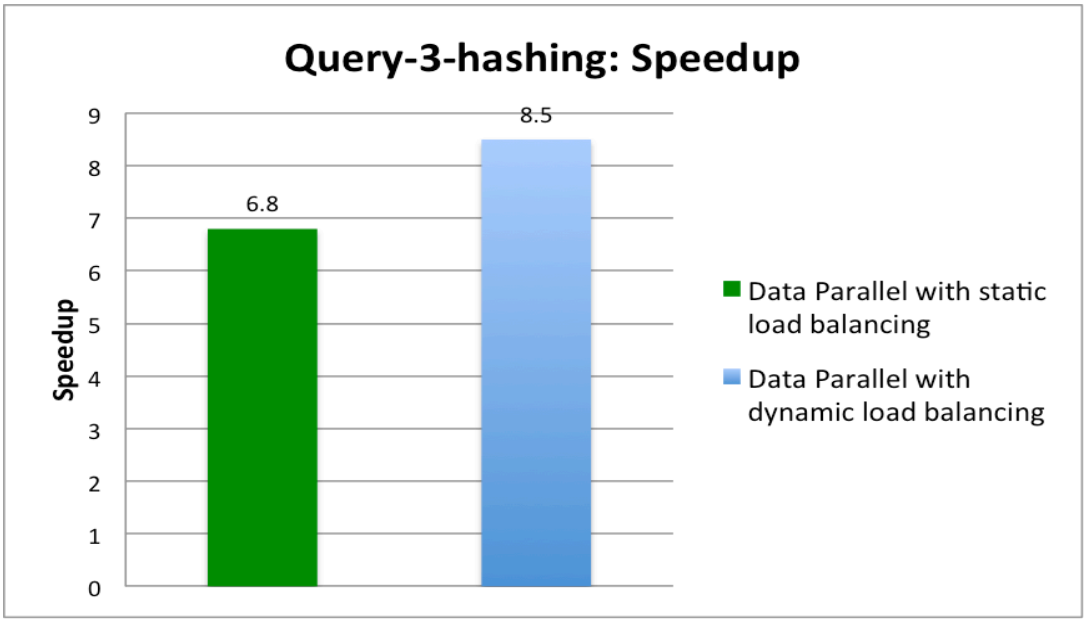
Το πιο πάνω μειονέκτημα αναιρείται μέσω του δυναμικού καταμερισμού εργασίας, αφού με την συγκεκριμένη προσέγγιση έχουμε καλύτερο και πιο δίκαιο διαμοιρασμό της εργασίας. Συγκεκριμένα, η επιτάχυνση για αυτή την προσέγγιση ανέρχεται στο 11,64 γεγονός που καθιστά την συγκεκριμένη προσέγγιση καλύτερη από την προηγούμενη.

6.1.3 Query-3-Hashing

Εδώ παρουσιάζουμε τα αποτελέσματα της εφαρμογής Query-3-Hashing. Ο χρόνος εκτέλεσης αυτής της εφαρμογής, είναι πολύ πιο χαμηλός από τον χρόνο εκτέλεσης της ίδιας εφαρμογής που δεν κάνει χρήση της τεχνικής του κατακερματισμού. Αυτό οφείλεται στο ότι η συγκεκριμένη τεχνική, μέσω της αποφυγής εκτέλεσης αχρείαστων ελέγχων, μειώνει τον αριθμό πράξεων και επαναλήψεων που εκτελούνται.
Στο σχήμα 6.3(α) παρατηρούμε τον χρόνο εκτέλεσης της συγκεκριμένης εφαρμογής, ενώ στο σχήμα 6.3(β) έχουμε την επιτάχυνση που επιτυγχάνεται με παραλληλισμό δεδομένων.



Σχήμα 6.3(α): *Χρόνος εκτέλεσης της εφαρμογής Query-3-hashing*



Σχήμα 6.3(β): *Επιτάχυνση για Query-3-hashing με παραλληλισμό δεδομένων*

Όπως παρατηρούμε από την γραφική παράσταση με τον χρόνο εκτέλεσης της εφαρμογής, με την χρήση παραλληλισμού δεδομένων επιτυγχάνουμε μεγάλη μείωση του χρόνου εκτέλεσης, τόσο με στατικό, τόσο και με δυναμικό καταμερισμό εργασίας. Σε καμία όμως περίπτωση, δεν έχουμε ιδεατή επιτάχυνση μέσω της χρήσης παραλληλισμού δεδομένων. Στον στατικό

καταμερισμό εργασίας, έχουμε το πρόβλημα του άνισου διαμοιρασμού δεδομένων και τον μεγαλύτερο φόρτο εργασίας σε κάποια νήματα παρά σε άλλα. Στον δυναμικό καταμερισμό εργασίας, ο διαμοιρασμός δεδομένων είναι ίσος και όλα τα νήματα έχουν σχεδόν ίδιο χρόνο εκτέλεσης, ο οποίος ισούται με 1,5 δευτερόλεπτα. Αυτό κανονικά μας δίνει επιτάχυνση 11,3. Όμως, ο υπόλοιπος χρόνος (0.5 δευτερόλεπτα) οφείλεται στην δημιουργία των νημάτων και στην επιστροφή της τιμής που υπολογίζουν. Αυτός είναι ο λόγος που η δυναμική υλοποίηση δεν έχει ιδεατή επιτάχυνση.

6.2 Παρουσίαση Αποτελεσμάτων για την Χρήση Διασωλήνωσης

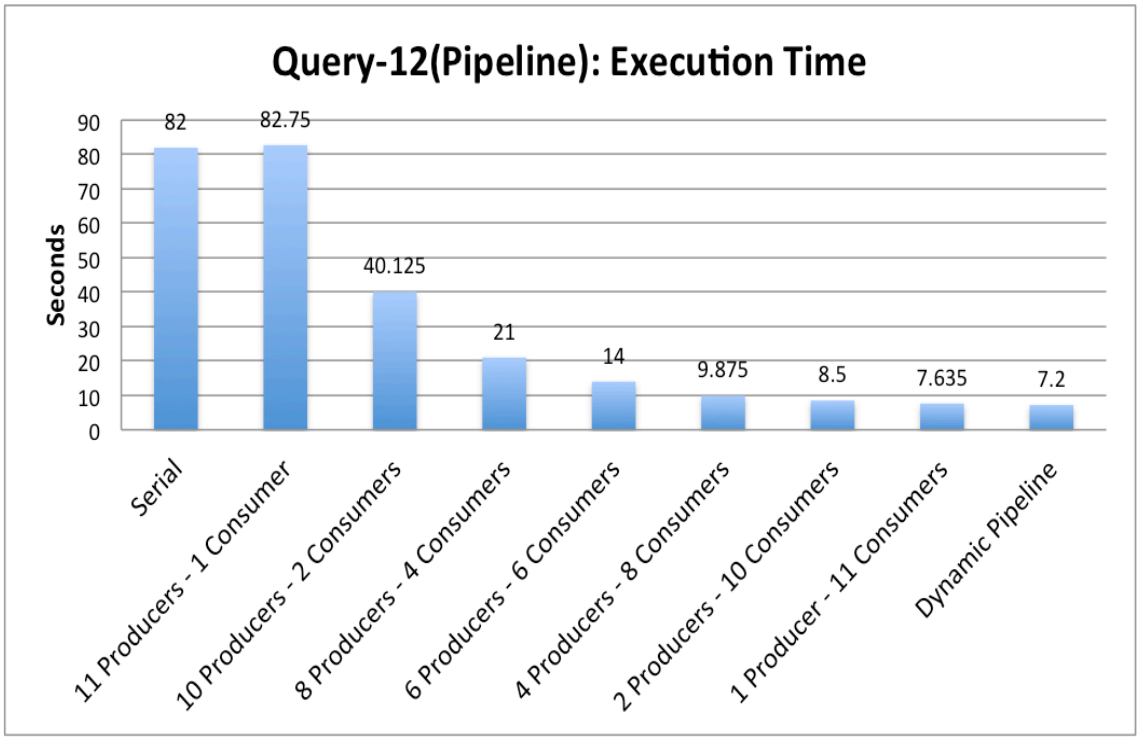
Σε αυτό το μέρος, παρουσιάζουμε τους χρόνους εκτέλεσης και την επιτάχυνση για τις εφαρμογές που χρησιμοποιήσαμε με την χρήση του στατικού και του δυναμικού αλγόριθμου διασωλήνωσης που περιγράψαμε στο κεφάλαιο 5. Τα νήματα που χρησιμοποιούνται σε κάθε πείραμα είναι 12, αφού τόσος είναι και ο αριθμός των πυρήνων της μηχανής στην οποία τρέχουμε τα πειράματα.
Στα πειράματα όπου χρησιμοποιήθηκε ο στατικός αλγόριθμος, έχουμε δοκιμάσει 7 διαφορετικούς συνδυασμούς με διαφορετικό αριθμό παραγωγών και καταναλωτών. Οι 7 διαφορετικοί συνδυασμοί, είναι οι εξής:

- 11 παραγωγοί, 1 καταναλωτής
- 10 παραγωγοί, 2 καταναλωτές
- 8 παραγωγοί, 4 καταναλωτές
- 6 παραγωγοί, 6 καταναλωτές
- 4 παραγωγοί, 8 καταναλωτές
- 2 παραγωγοί, 10 καταναλωτές
- 1 παραγωγός, 11 καταναλωτές

Ο λόγος που δοκιμάσαμε όλους αυτούς τους συνδυασμούς, είναι για να εξετάσουμε πώς επηρεάζεται η επίδοση του στατικού αλγορίθμου με διαφορετικό αριθμό παραγωγών - καταναλωτών και γιατί.

6.2.1 Query-12

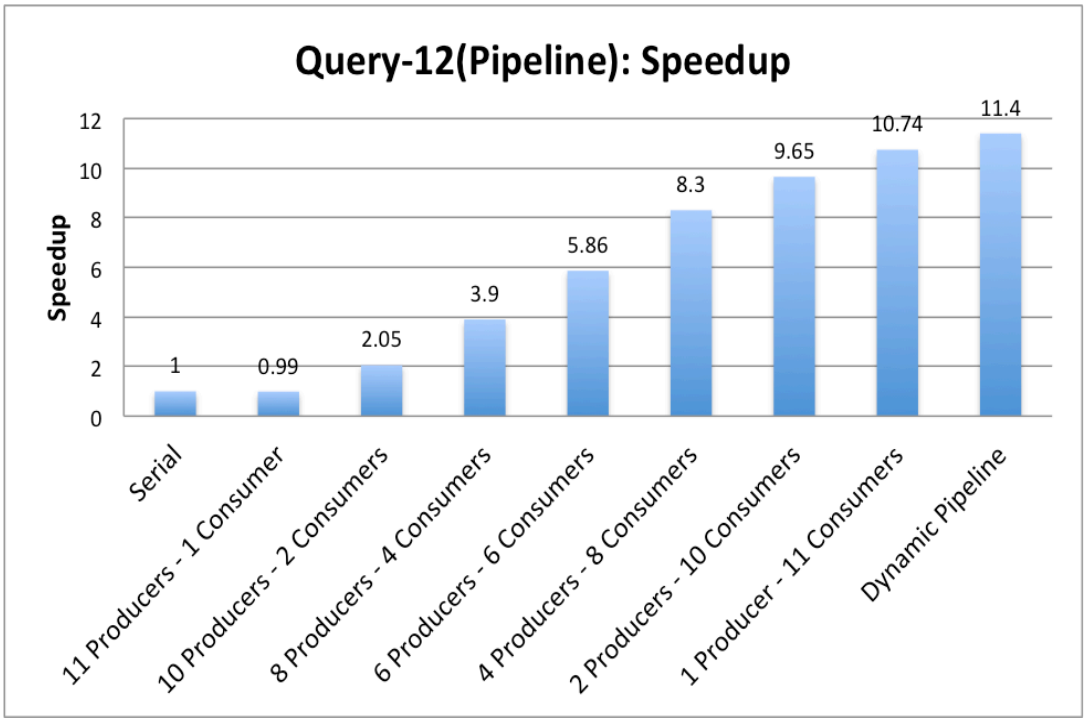
Η συγκεκριμένη εφαρμογή, αποτελείται από μια λειτουργία σάρωσης και μία λειτουργία σύνδεσης. Αυτό που κάναμε λοιπόν, ήταν να αναθέσουμε την εκτέλεση της σάρωσης στους παραγωγούς και την εκτέλεση της σύνδεσης στους καταναλωτές.
Ο χρόνος εκτέλεσης για όλους τους συνδυασμούς με τη χρήση στατικής διασωλήνωσης φαίνεται στο σχήμα 6.4(α).



Σχήμα 6.4(α): *Χρόνος εκτέλεσης δυναμικής διασωλήνωσης και στατικής διασωλήνωσης για όλους τους συνδυασμούς παραγωγών – καταναλωτών*

Όπως παρατηρούμε απο τον χρόνο εκτέλεσης για όλους τους συνδυασμούς όπου έχουμε στατικό καταμερισμό εργασίας, ο χρόνος εκτέλεσης μειώνεται όσο αυξάνεται ο αριθμός των καταναλωτών, με αποτέλεσμα ο καλύτερος συνδυασμός να είναι αυτός με τον 1 παραγωγό και τους 11 καταναλωτές. Αυτό συμβαίνει επειδή η δουλειά των παραγωγών είναι πιο λίγη, με αποτέλεσμα να είναι πιο γρήγοροι απο ότι οι καταναλωτές. Για αυτό τον λόγο στην

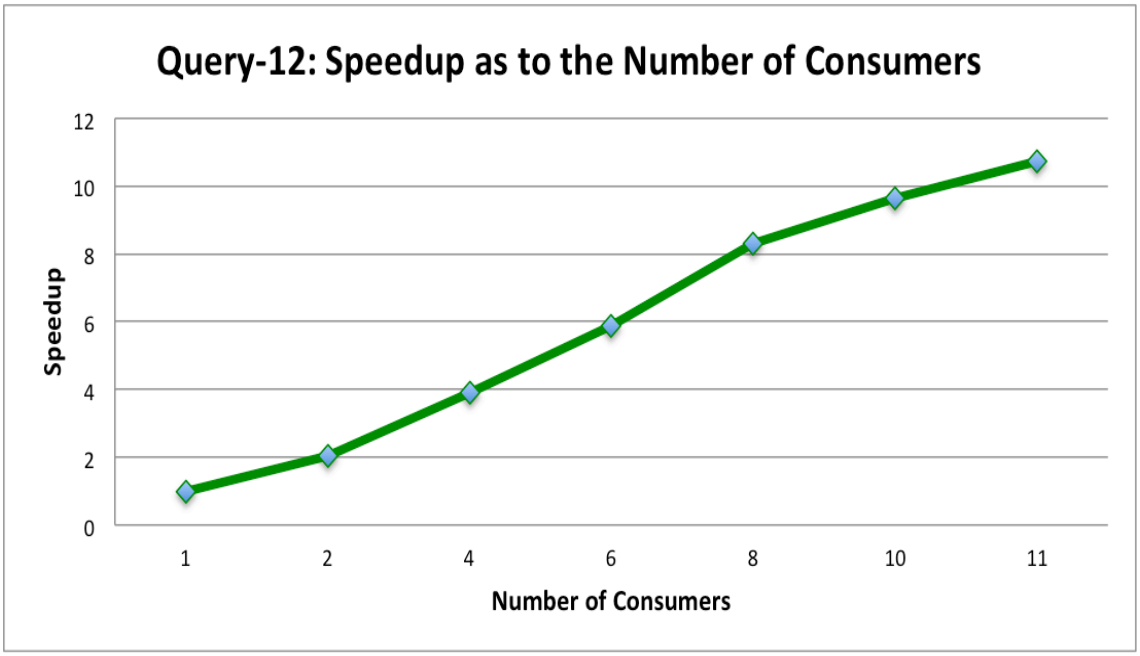
συγκεκριμένη εφαρμογή, όσο πιο πολλοί είναι οι καταναλωτές, τόσο πιο πολλή επίδοση έχουμε. Επίσης, παρατηρούμε ότι με δυναμική διασωλήνωση έχουμε λιγότερο χρόνο εκτέλεσης από ότι ο καλύτερος χρόνος εκτέλεσης που επιτυγχάνεται με το στατική διασωλήνωση. Ο λόγος που συμβαίνει αυτό, είναι γιατί εκμεταλλεύεται καλύτερα τον παραλληλισμό αφού μόλις τελειώσουν όλοι οι παραγωγοί (που όπως είπαμε πιο πάνω είναι πιο γρήγοροι απ' τους καταναλωτές σ' αυτή την εφαρμογή), γίνονται κι αυτοί καταναλωτές με αποτέλεσμα σε κάποια φάση της εκτέλεσης να έχουμε 12 καταναλωτές, σε αντίθεση με την στατική διασωλήνωση όπου για 1 παραγωγό και 11 καταναλωτές ο αριθμός των καταναλωτών παραμένει 11 καθ'όλη την διάρκεια της εκτέλεσης, ακόμα και αν ο ένας παραγωγός τελειώνει πολύ πιο γρήγορα την δική του εργασία.



Σχήμα 6.4(β): *Επιτάχυνση για το Query-12 με χρήση διασωλήνωσης*

Στο σχήμα 6.4(β) έχουμε την γραφική παράσταση για την επιτάχυνση που επιτυγχάνεται για κάθε συνδυασμό, ενώ στο σχήμα 6.4(γ) μπορούμε να δούμε τον ρυθμό αύξησης της επιτάχυνσης σε αναλογία με τον αριθμό των καταναλωτών που ορίζουμε στην διασωλήνωση. Η επιτάχυνση για την

συγκεκριμένη εφαρμογή, είναι ανάλογη του αριθμού των καταναλωτών και αυτό συμβαίνει για τον λόγο που εξηγήσαμε και πιο πάνω, ότι δηλαδή οι καταναλωτές είναι πιο αργοί από τους παραγωγούς. Αυτό που μπορούμε να εξάγουμε ώς συμπέρασμα από τα αποτελέσματα για αυτή την εφαρμογή, είναι ότι ο αριθμός των καταναλωτών και των παραγωγών που ορίζουμε, παίζει καθοριστικό ρόλο στην επίδοση της εφαρμογής. Άλλη σημαντική παρατήρηση, είναι ότι η δυναμική διασωλήνωση υπερέχει της στατικής, αφού με αυτήν επιτυγχάνουμε επιτάχυνση πολύ κοντά στην ιδεατή.



Σχήμα 6.4(γ): *Επιτάχυνση ώς προς τον αριθμό των καταναλωτών για το Query-12 με την χρήση στατικής διασωλήνωσης*

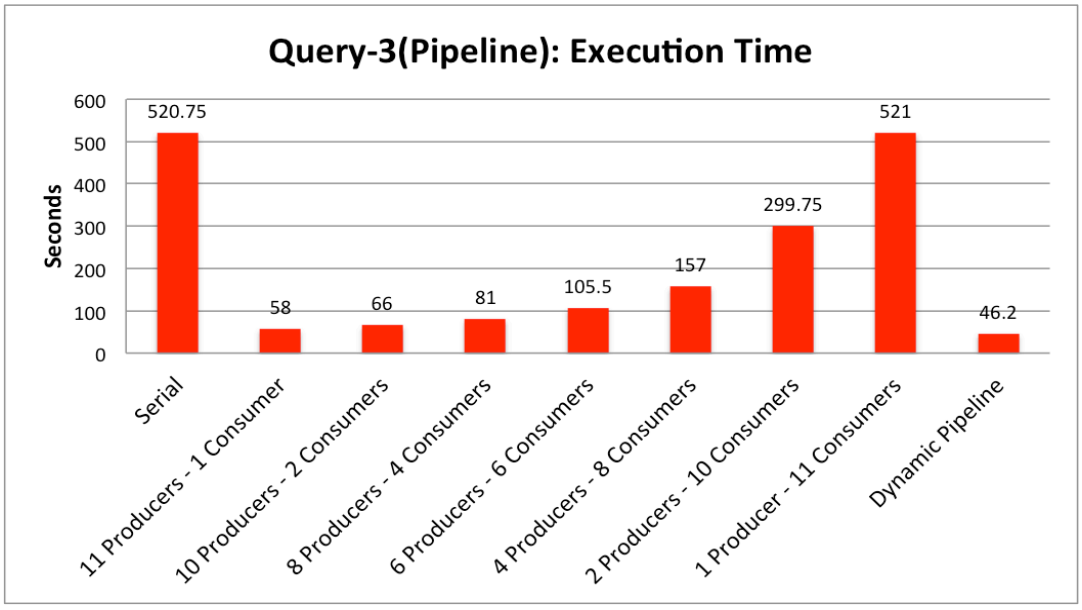
6.2.2 Query-3

Η συγκεκριμένη εφαρμογή, αποτελείται απο 3 λειτουργίες σάρωσης και 2 λειτουργίες σύνδεσης. Στους παραγωγούς δίνουμε τις 2 λειτουργίες σάρωσης και την μία λειτουργία σύνδεσης, ενώ στους καταναλωτές δίνουμε την εκτέλεση της 3^{ης} σάρωσης και της 2^{ης} λειτουργίας σύνδεσης.

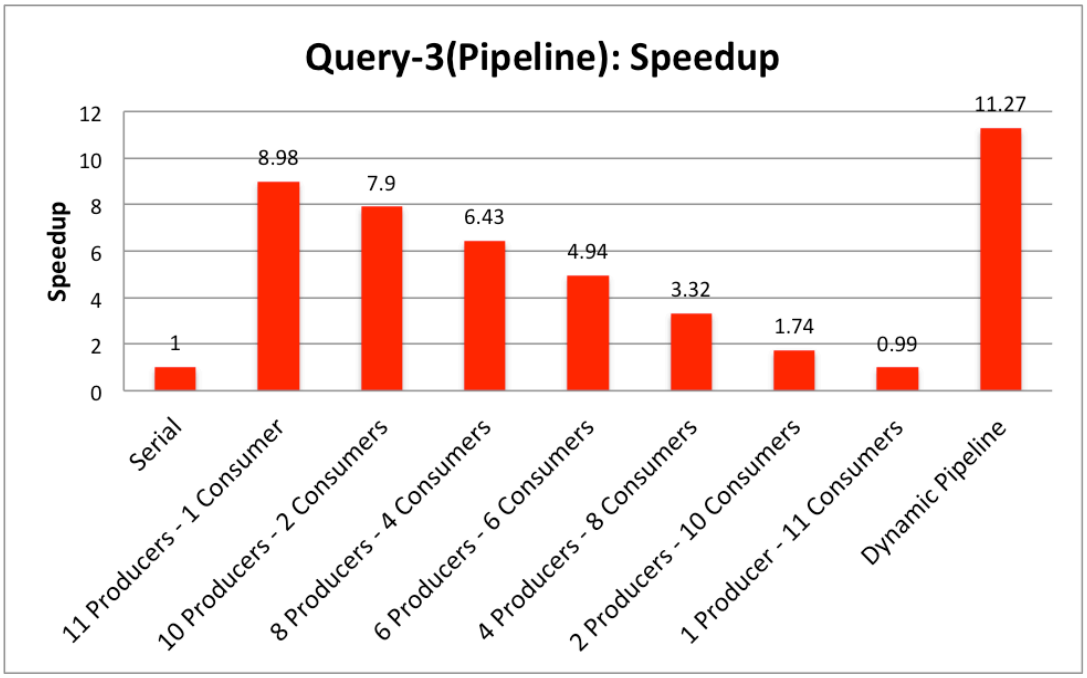
Στο σχήμα 6.5(α) παρουσιάζουμε τον χρόνο εκτέλεσης για κάθε ξεχωριστό συνδυασμό που χρησιμοποιούμε στην στατική διασωλήνωση για το Query-3, αλλά και το χρόνο εκτέλεσης με την χρήση δυναμικής διασωλήνωσης.
Όπως παρατηρούμε, όπως και στο Query-12, ο χρόνος εκτέλεσης με τη χρήση δυναμικής διασωλήνωσης, είναι καλύτερος απο όποιοδήποτε άλλο συνδυασμό της στατικής διασωλήνωσης. Επίσης, ο χρόνος εκτέλεσης στη στατική διασωλήνωση μειώνεται όσο αυξάνεται ο αριθμός των παραγωγών στην συγκεκριμένη εφαρμογή, πράγμα που έρχεται σε αντίθεση με την μείωση του χρόνου στο Query-12 όσο αυξάναμε τον αριθμό των καταναλωτών. Αυτό συμβαίνει λόγω του ότι σε αυτή την εφαρμογή, οι καταναλωτές είναι πιο γρήγοροι από τους παραγωγούς, με αποτέλεσμα πολλές φορές να μένουν χωρίς εργασία περιμένοντας τους παραγωγούς να τοποθετήσουν κάποιο αποτέλεσμά τους στον κοινόχρηστο πίνακα. Αυτό το έχουμε συμπεράνει απο μετρικές που έχουμε βάλει στον κώδικα των καταναλωτών, για να δούμε πόσες φορές προσπαθούν να διαβάσουν από τον πίνακα αλλά δεν έχει έτοιμο αποτέλεσμα. Όσο αυξάνουμε τον αριθμό των παραγωγών, παρατηρούμε ότι μειώνεται ο αριθμός που ένας καταναλωτής παραμένει χωρίς εργασία και αυτός είναι ο λόγος που έχουμε μεγαλύτερη επίδοση κάθε φορά που αυξάνουμε τον αριθμό των παραγωγών και μειώνουμε αυτό των καταναλωτών.
Αυτό φαίνεται και από την γραφική παράσταση του σχήματος 6.5(β), αλλά και απο την γραφική παράσταση του σχήματος 6.5(γ) όπου παρατηρούμε καθοδική πορεία της επιτάχυνσης όσο αυξάνεται ο αριθμός των καταναλωτών.

Απο την γραφική παράσταση του σχήματος 6.5(β), μπορούμε επίσης να προσέξουμε την σημαντικότητα που έχει στην αύξηση της επίδοσης η χρήση του δυναμικού αλγόριθμου διασωλήνωσης που έχουμε υλοποιήσει, αφού η επιτάχυνση αγγίζει την ιδεατή. Ο λόγος για τον οποίο έχουμε πιο ψηλή επιτάχυνση με δυναμική διασωλήνωση απο ότι με τον καλύτερο συνδυασμό στατικής διασωλήνωσης (11 παραγωγοί – 1 καταναλωτής), είναι λόγω του ότι ο χρόνος που χρειάζονται για την εκτέλεση τους οι παραγωγοί στην δυναμική υλοποίηση, είναι μικρότερος απο τον χρόνο εκτέλεσης στην στατική υλοποίηση. Αυτό συμβαίνει διότι στην στατική υλοποίηση οι παραγωγοί μοιράζονται τα δεδομένα που επεξεργάζονται στατικά, έχοντας έτσι τα προβλήματα του

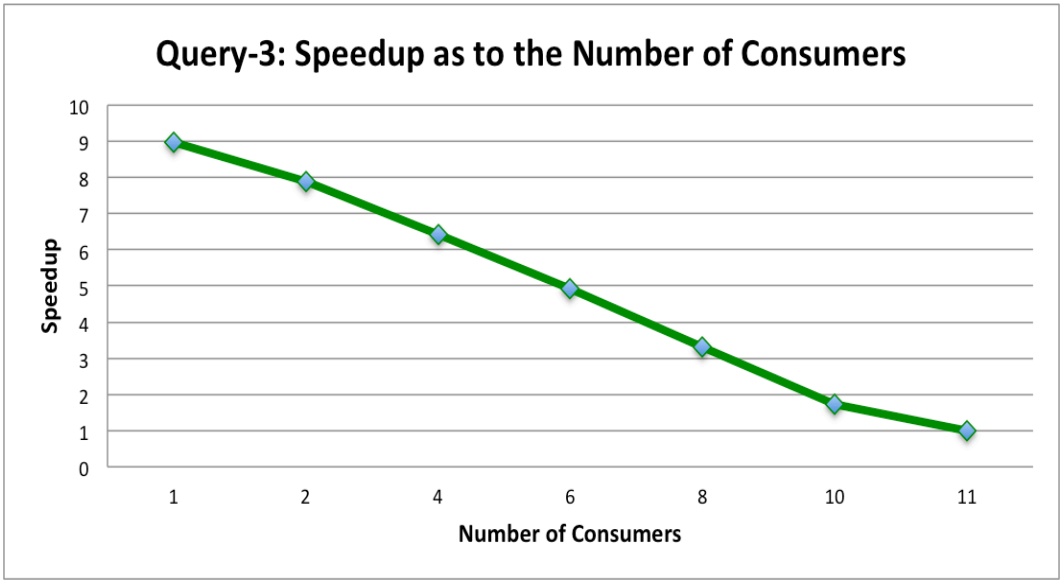
στατικού καταμερισμού εργασίας που περιγράψαμε σε προηγούμενο υποκεφάλαιο. Αντίθετα, στην δυναμική διασωλήνωση, οι παραγωγοί μοιράζονται τα δεδομένα δυναμικά, με αποτέλεσμα ο καταμερισμός εργασίας μεταξύ τους να είναι πιο δίκαιος και πιο σωστός, αφού αν κάποιος παραγωγός είναι πιο γρήγορος απο κάποιον άλλο θα επεξεργαστεί περισσότερα δεδομένα.



Σχήμα 6.5(α): *Χρόνος εκτέλεσης για το Query-3 με χρήση διασωλήνωσης*

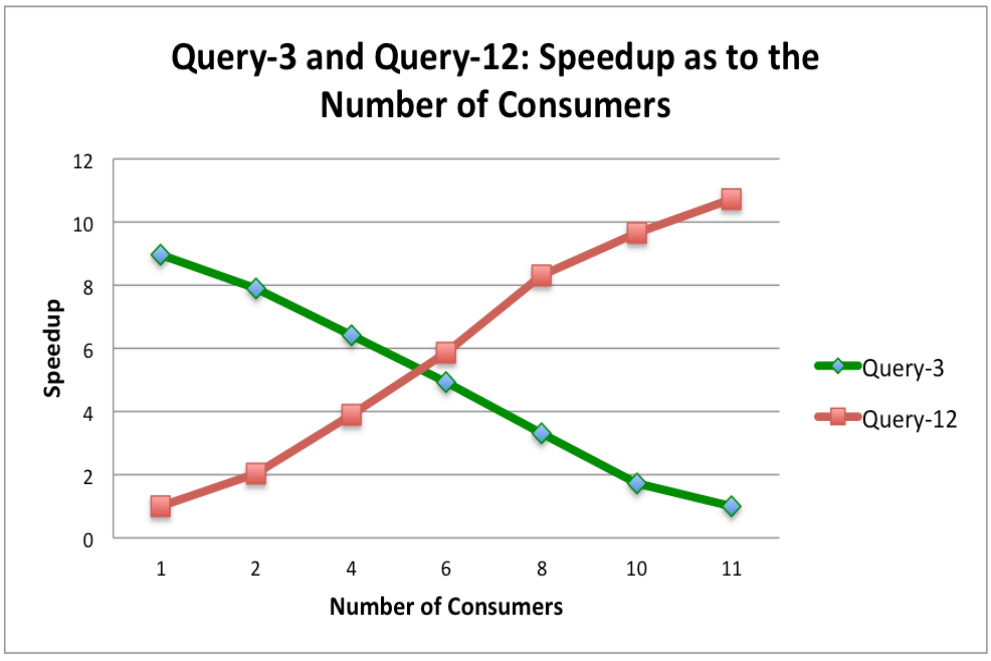


Σχήμα 6.5(β): *Επιτάχυνση για το Query-3 με χρήση διασωλήνωσης*



Σχήμα 6.5(γ): *Επιτάχυνση ώς προς τον αριθμό των καταναλωτών για το Query-3 με την χρήση στατικής διασωλήνωσης*

Σε αυτό το σημείο, είναι σημαντικό να δούμε την σημαντικότητα και σπουδαιότητα του δυναμικού αλγόριθμου έναντι του στατικού αλγορίθμου. Ας παρατηρήσουμε την γραφική παράσταση του σχήματος 6.6.



Σχήμα 6.6: *Επιτάχυνση ώς προς τον αριθμό των καταναλωτών για τα Queries 3 και 12 με χρήση στατικής διασωλήνωσης*

Όπως παρατηρούμε, για το Query-3 συμφέρει να έχουμε όσο πιο λιγότερο αριθμό καταναλωτών, ενώ για το Query-12 το αντίθετο. Πώς όμως μπορεί να καταλήξει κανείς στον πιο καλό συνδυασμό παραγωγών – καταναλωτών; Πρέπει να μελετήσει ποια ομάδα (παραγωγοί ή καταναλωτές) εκτελεί περισσότερη εργασία απο την άλλη και αναλόγως να μοιράσει τα νήματα κατάλληλα. Στον δυναμικό όμως αλγόριθμο, δεν χρειάζεται αυτή η μελέτη, αφού η εκτέλεση ξεκινά με ίσο αριθμό απο παραγωγούς και καταναλωτές και στην συνέχεια ο αριθμός των νημάτων μίας ομάδας αυξάνεται (και συνεπώς μειώνεται ο αριθμός της άλλης ομάδας) εάν αυτή η ομάδα έχει περισσότερη δουλειά απο την άλλη. Αυτό, είναι ένα μεγάλο πλεονέκτημα που παρέχεται από τον δυναμικό αλγόριθμο. Επίσης, ο δυναμικός αλγόριθμος εκμεταλλεύεται καλύτερα τον παραλληλισμό που παρέχεται από το σύστημα και έτσι έχει μεγαλύτερη επιτάχυνση για τις εφαρμογές που έχουμε δεί πιο πάνω. Ο λόγος για τον οποίο εκμεταλλεύεται καλύτερα τον παραλληλισμό, έχει εξηγηθεί πιο πάνω.

6.2.3 Query-3-Hashing

Οι λειτουργίες σάρωσης για αυτή την εφαρμογή εκτελούνται προτού αρχίσει η διαδικασία για υπολογισμό του αποτελέσματος και τα δεδομένα ενός πίνακα αποθηκεύονται στον ανάλογο πίνακα κατακερματισμού. Οπόταν, σε αυτή την εφαρμογή έχουμε μόνο τις 2 λειτουργίες σύνδεσης, εκ των οποίων η μία εκτελείται από τους παραγωγούς και η άλλη από τους καταναλωτές.

Στην συγκεκριμένη εφαρμογή, δυστυχώς, δεν έχουμε καταφέρει την επίτευξη επιτάχυνσης με την χρήση διασωλήνωσης. Αρχικά, δοκιμάσαμε να τρέξουμε την εφαρμογή με τον συνδυασμό με 6 παραγωγούς και 6 καταναλωτές μέσω στατικής διασωλήνωσης. Αυτό, είχε χρόνο εκτέλεσης 29 δευτερόλεπτα, δηλαδή 12 δευτερόλεπτα χειρότερο από τον σειριακό χρόνο εκτέλεσης. Χρησιμοποιήσαμε μετρικές εντός του κώδικα για να δούμε ποιά ομάδα είναι πιο αργή και ανακαλύψαμε ότι αυτή η ομάδα είναι οι παραγωγοί, αφού οι καταναλωτές έμεναν πολλές φορές χωρίς εργασία περιμένοντας τους παραγωγούς να παράξουν κάποιο αποτέλεσμα. Επομένως, αυξήσαμε τον αριθμό των παραγωγών απο 6 σε 8 και μειώσαμε αυτό των καταναλωτών κατά 2 (δηλαδή έχουμε 8 παραγωγούς και 4 καταναλωτές). Ο χρόνος εκτέλεσης

μειώθηκε από τα 29 στα 21 δευτερόλεπτα, όμως και πάλι είναι χειρότερος από τον σειριακό χρόνο εκτέλεσης. Με μετρικές που χρησιμοποιήσαμε και πάλι εντός του κώδικα, παρατηρήσαμε ότι η πιο αργή ομάδα αυτή την φορά είναι οι καταναλωτές, αφού όταν τελειώνουν οι παραγωγοί, οι καταναλωτές έχουν καταναλώσει μόνο μικρό μέρος της συνολικής εργασίας τους. Συνεπώς, θα ήταν άστοχο εάν δοκιμάζαμε να τρέξουμε την εφαρμογή με περισσότερους απο 8 παραγωγούς, όπως το ίδιο άστοχο θα ήταν και αν δοκιμάζαμε να τρέξουμε την εφαρμογή με λιγότερους απο 6 παραγωγούς (αφού για ίσο αριθμό παραγωγών – καταναλωτών οι καταναλωτές είναι πιο γρήγοροι). Στον πίνακα 6.1 έχουμε τον σειριακό χρόνο εκτέλεσης για την εφαρμογή Query-3-Hashing και τον χρόνο εκτέλεσης για την ίδια εφαρμογή με την χρήση διασωλήνωσης.

	Serial	**6 producers – 6 consumers**	**8 producers – 4 consumers**
Query-3-Hashing	17 δευτερόλεπτα	29 δευτερόλεπτα	21 δευτερόλεπτα

Πίνακας 6.1: *Χρόνος εκτέλεσης με την χρήση διασωλήνωσης*

Για να βρούμε ποιο κομμάτι του κώδικα των παραγωγών και των καταναλωτών έχει την μεγαλύτερη καθυστέρηση, μετρήσαμε τον χρόνο ανεξάρτητων κομματιών. Ανακαλύψαμε πώς, οι μεγαλύτερες καθυστερήσεις οφείλονται λόγω των λειτουργιών γραψίματος σε συνεχόμενες θέσεις μνήμης.

Μία τέτοια καθυστέρηση είναι η αύξηση του σημαφόρου ενός παραγωγού, αφού οι σημαφόροι των παραγωγών είναι αποθηκευμένοι σε **συνεχόμενες θέσεις** ενός κοινόχρηστου πίνακα με αριθμό θέσεων όσος και ο αριθμός των παραγωγών (μία θέση για κάθε σημαφόρο). Πιο συγκεκριμένα, αυτή η καθυστέρηση είναι 12 δευτερόλεπτα και αποτελεί το 57% του συνολικού χρόνου εκτέλεσης του προγράμματος (12 από τα 21 συνολικά δευτερόλεπτα). Επίσης, απο μόνη της η καθυστέρηση αυτή είναι έξι φορές μεγαλύτερη απο ότι ο χρόνος εκτέλεσης όλων των υπολογισμών με την χρήση παραλληλισμού δεδομένων. Για σκοπούς περαιτέρω ανάλυσης αυτής της καθυστέρησης, αναθέσαμε σε κάθε παραγωγό ένα τοπικό σημαφόρο-μετρητή, για να ελέγξουμε την διαφορά

στον χρόνο εκτέλεσης απο ότι όταν ο σημαφόρος είναι καθολικός (global). Ο χρόνος εκτέλεσης μόνο για την αύξηση του τοπικού σημαφόρου ενός παραγωγού, είναι 2 δευτερόλεπτα, δηλαδή έξι φορές μικρότερος απο ότι όταν ο σημαφόρος είναι καθολικός. Στον αλγόριθμό μας όμως, είναι αναγκαίο ο σημαφόρος κάθε παραγωγού να είναι καθολικός, ώστε να μπορούν οι καταναλωτές που αντιστοιχούν σε ένα παραγωγό να ελέγχουν την τιμή του για να υπάρχει ο αναγκαίος συγχρονισμός.
Παρόμοια καθυστέρηση με την πιο πάνω έχουμε και με την αύξηση του σημαφόρου ενός καταναλωτή, αφού στην στατική υλοποίηση οι σημαφόροι των καταναλωτών είναι αποθηκευμένοι σε συνεχόμενες θέσεις ενός κοινόχρηστου πίνακα όπως και αυτοί των παραγωγών.
Οι πιο πάνω καθυστερήσεις, αποτελούν στενώσεις στην αύξηση της επίδοσης της εφαρμογής αυτής, αλλά και πιθανόν άλλων τέτοιων εφαρμογών οι οποίες χρησιμοποιούν τεράστιο όγκο δεδομένων. Αυτός είναι και ο κύριος λόγος για τον οποίο η συγκεκριμένη εφαρμογή, με τον τεράστιο όγκο δεδομένων που της δίνουμε ώς είσοδο, είναι πολύ χειρότερη με τη χρήση στατικής διασωλήνωσης από την ανάλογη εφαρμογή με παραλληλισμό δεδομένων.

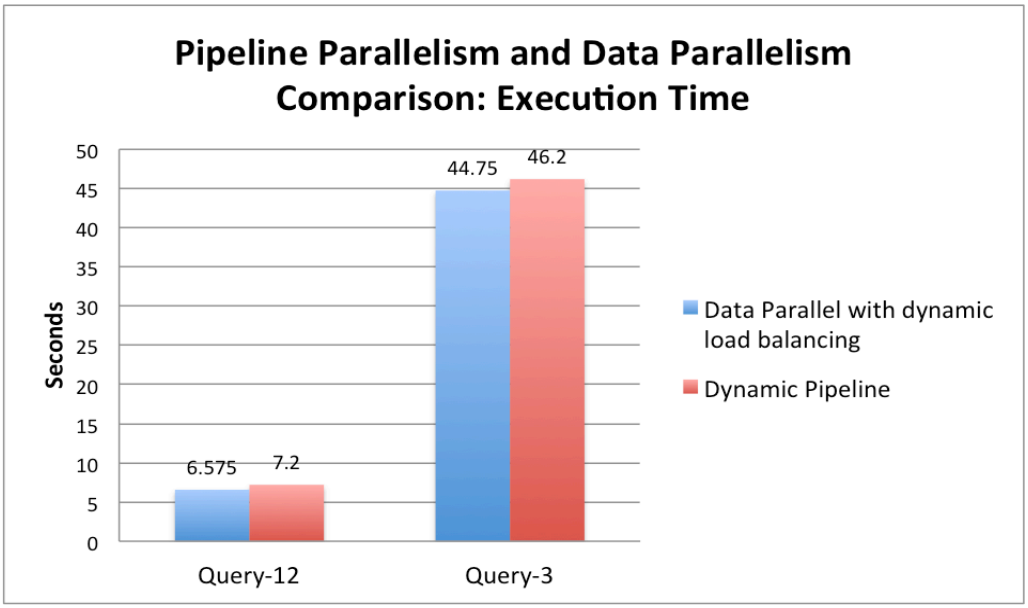
Στην συνέχεια, τρέξαμε αυτή την εφαρμογή με τη χρήση του αλγορίθμου δυναμικής διασωλήνωσης που υλοποιήσαμε. Ο χρόνος εκτέλεσης, είναι πολύ χειρότερος και μεγαλύτερος απο αυτόν της στατικής διασωλήνωσης. Ο λόγος για τον οποίο συμβαίνει αυτό, είναι ότι εκτός απο τις καθυστερήσεις – στενώσεις που έχουμε απο τις λειτουργίες γραψίματος σε συνεχόμενες θέσεις μνήμης (λόγω του κοινόχρηστου πίνακα αυτή την φορά), υπάρχει και αυστηρό κλείδωμα (locking) των διαφόρων κοινόχρηστων μεταβλητών, για να μπορεί να έχει σωστό αποτέλεσμα η εφαρμογή.

Συνεπώς, όπως βλέπουμε, η διασωλήνωση που έχουμε υλοποιήσει έχει το σοβαρό μειονέκτημα του ότι τα νήματα γράφουν σε συνεχόμενες θέσεις στην μνήμη (αναγκαστικά λόγω επικοινωνίας που πρέπει να υφίσταται μεταξύ των νημάτων), το οποίο έχει ώς αποτέλεσμα πολλές καθυστερήσεις στον χρόνο εκτέλεσης. Εκτός απο αυτό το μειονέκτημα, η υλοποίησή μας πάσχει και απο το

αυστηρό κλείδωμα των κοινόχρηστων μεταβλητών, που και αυτό έχει αρνητικό αντίκτυπο στον χρόνο εκτέλεσης.

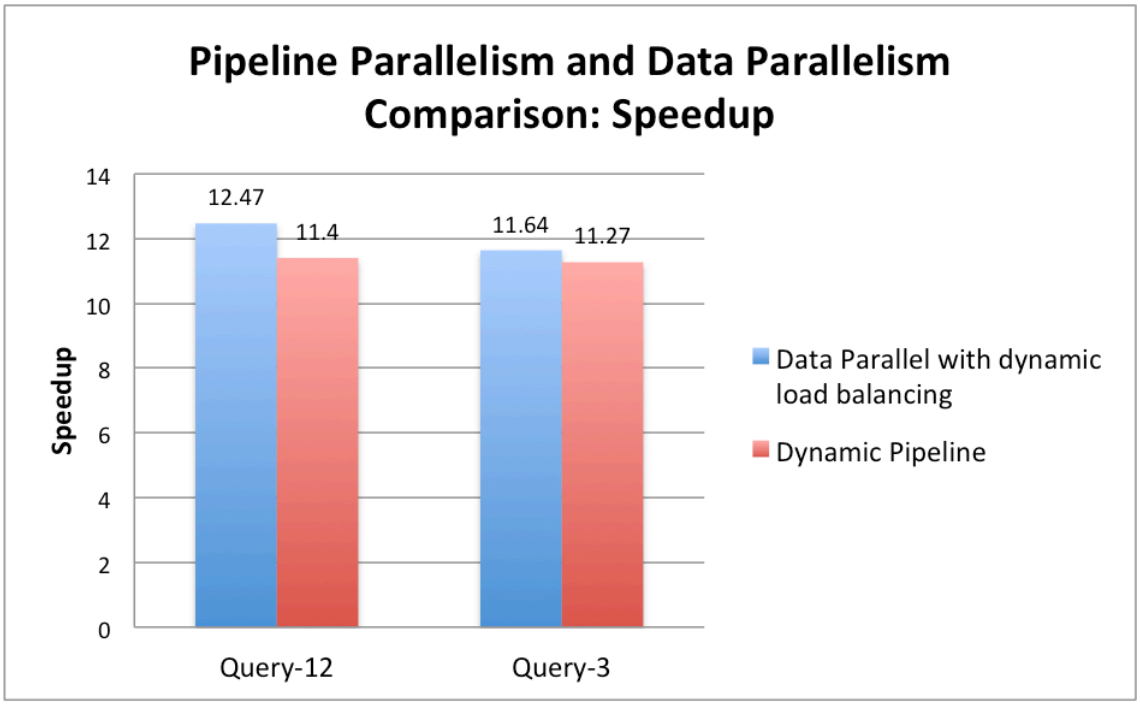
6.3 Σύγκριση των Δύο Μορφών Παραλληλισμού

Στα προηγούμενα υποκεφάλαια αυτού του κεφαλαίου, παρουσιάσαμε τα αποτελέσματα που έχουμε για τις δύο μορφές παραλληλισμού τις οποίες είχαμε ώς στόχο να συγκρίνουμε με την διεκπεραίωση αυτής της διπλωματικής εργασίας: τον παραλληλισμό δεδομένων και τον παραλληλισμό με διασωλήνωση. Στην εφαρμογή Query-3-Hashing, είδαμε πώς τόσο ο στατικός αλγόριθμος όσο και ο δυναμικός αλγόριθμος που κάνουν χρήση διασωλήνωσης, έχουν κακή επίδοση σε αντίθεση με τον παραλληλισμό δεδομένων με δυναμικό καταμερισμό εργασίας που επιτυγχάνει όχι ιδεατή αλλά αρκετά καλή επιτάχυνση. Όσο αφορά τις άλλες δύο εφαρμογές, παρατηρήσαμε ότι οι δυναμικές μας υλοποιήσεις, τόσο με παραλληλισμό δεδομένων, τόσο και με χρήση διασωλήνωσης είναι πολύ καλύτερες από τις αντίστοιχες στατικές υλοποιήσεις. Για αυτό τον λόγο, στο συγκεκριμένο κεφάλαιο συγκρίνουμε τις καλύτερες υλοποιήσεις απο την κάθε μορφή παραλληλισμού. Δηλαδή, συγκρίνουμε τα αποτελέσματα που έχουμε για τον παραλληλισμό δεδομένων με δυναμικό καταμερισμό εργασίας, με τα αποτελέσματα που έχουμε με την χρήση δυναμικής διασωλήνωσης.



Σχήμα 6.7(α): *Χρόνος εκτέλεσης για τις δύο μορφές παραλληλισμού*

Όπως παρατηρούμε από την γραφική παράσταση του σχήματος 6.7(α) και για τις δύο εφαρμογές (Query-12 και Query-3), η υλοποίηση με παραλληλισμό δεδομένων είναι αυτή που έχει τον πιο λίγο χρόνο εκτέλεσης και συνεπώς και μεγαλύτερη επιτάχυνση, όπως παρατηρούμε στην γραφική παράσταση του σχήματος 6.7(β).



Σχήμα 6.7(β): *Επιτάχυνση για τις δύο μορφές παραλληλισμού*

Το βασικό συμπέρασμα στο οποίο καταλήγουμε λοιπόν, είναι ότι όσο αφορά τις εφαρμογές που έχουμε χρησιμοποιήσει για την αξιολόγηση των δύο μορφών παραλληλισμού, ο παραλληλισμός δεδομένων παράγει καλύτερα αποτελέσματα απο ότι οι αλγόριθμοι που έχουμε υλοποιήσει για στατική και δυναμική διασωλήνωση. Ο κυριότερος λόγος για τον οποίο συμβαίνει αυτό, είναι ότι μέσω του παραλληλισμού δεδομένων δέν χρειάζεται ο συγχρονισμός μεταξύ των νημάτων με κλειδαριές (locks) ή η αναμονή όταν κάποιο νήμα είναι πιο αργό απο τα άλλα (κάτι το οποίο είναι απαραίτητο στην διασωλήνωση).

Συγκεκριμένα, στον αλγόριθμο στατικής διασωλήνωσης, έχουμε απο την μιά το πρόβλημα της αναμονής των καταναλωτών, όταν είναι πιο αργοί οι παραγωγοί,

ενώ απο την άλλη έχουμε το πρόβλημα της εκτέλεσης του μεγαλύτερου μέρους της εργασίας από τους καταναλωτές στην περίπτωση που η δουλειά των παραγωγών είναι πιο λίγη. Την λύση σε αυτό το πρόβλημα έδωσε όπως είδαμε ο αλγόριθμος δυναμικής διασωλήνωσης που υλοποιήσαμε, όμως ούτε αυτός είναι καλύτερος από τον παραλληλισμό δεδομένων (με δυναμικό καταμερισμό εργασίας), αφού στην υλοποίησή μας έχουμε αναγκαίο αυστηρό συγχρονισμό μεταξύ των νημάτων για να μήν έχουμε οποιοδήποτε λάθος στο αποτέλεσμα εξόδου των εφαρμογών. Ο συγκεκριμένος συγχρονισμός, αποτελεί στένωση για τον χρόνο εκτέλεσης, αφού το overhead του κυμαίνεται από 700 έως 800 milliseconds, μετά απο μετρήσεις που κάναμε στον κώδικα των δύο πιο πάνω εφαρμογών. Αυτός είναι και ο κύριος λόγος για τον οποίο η επίδοση με χρήση δυναμικής διασωλήνωσης είναι πιο λίγη απο την επίδοση με παραλληλισμό δεδομένων.

Κεφάλαιο 7

Συμπεράσματα

7.1 Συμπεράσματα

Στην εργασία αυτή, προσπαθήσαμε όσο μπορέσαμε να εκπληρώσουμε τον στόχο μας, ο οποίος είναι η βελτίωση εφαρμογών Βάσεων Δεδομένων με την χρήση παραλληλισμού δεδομένων και παραλληλισμού με διασωλήνωση, αλλά και η σύγκριση των δύο αυτών μορφών παραλληλισμού. Για την εκπλήρωση του σκοπού μας, απαραίτητό ήταν πρώτα να βρούμε εφαρμογές διαφορετικών κατηγοριών, ώστε να είναι όσο πιο σωστή η αξιολόγηση των δύο μορφών παραλληλισμού που θέλαμε να συγκρίνουμε. Έτσι λοιπόν, χρησιμοποιήσαμε 2 εφαρμογές από το “TPC-H Benchmark Suite” (την μία εκ των οποίων υλοποιήσαμε και με την χρήση κατακερματισμού), οι οποίες διαφέρουν στα μεταξύ τους χαρακτηριστικά.

Για την αξιολόγηση λοιπόν του παραλληλισμού με την χρήση διασωλήνωσης, αναπτύξαμε δύο δικούς μας αλγορίθμους, εκ των οποίων ο ένας χρησιμοποιά διασωλήνωση με στατικό καταμερισμό εργασίας, ενώ ο άλλος κάνει χρήση δυναμικού καταμερισμού εργασίας. Αναπτύξαμε επίσης και τις ανάλογες υλοποιήσεις που κάνουν χρήση παραλληλισμού δεδομένων.

Ένα συμπέρασμα στο οποίο καταλήξαμε, είναι το ότι όταν χρησιμοποιείται οποιοσδήποτε στατικός αλγόριθμος διασωλήνωσης, πρέπει να ορίζεται κατάλληλα ο αριθμός των παραγωγών και των καταναλωτών, αφού όπως

είδαμε από τα πειράματά μας είναι κάτι που μπορεί να παίξει καθοριστικό ρόλο στην επίδοση μιας εφαρμογής. Πιο συγκεκριμένα, πρέπει να δίνεται μεγαλύτερος αριθμός απο νήματα στην ομάδα που έχει την πιο μεγάλη εργασία.

Άλλο βασικό συμπέρασμα στο οποίο καταλήξαμε, είναι ότι οι δύο αλγόριθμοι με διασωλήνωση που υλοποιήσαμε έχουν ψηλή επίδοση για μικρά μεγέθη αρχείων εισόδου. Επίσης, καταλήξαμε στο ότι η χρήση δυναμικής διασωλήνωσης, έχει ψηλότερη επίδοση απο ότι η χρήση της στατικής διασωλήνωσης για αρχεία εισόδου μικρού σχετικά μεγέθους. Αυτό συμβαίνει λόγω του ότι με τη χρήση του αλγόριθμου δυναμικής διασωλήνωσης, κατά την διάρκεια εκτέλεσης μιας εφαρμογής, αλλάζει ο αριθμός των παραγωγών και των καταναλωτών ώστε να ενισχύεται περισσότερο η πιο αργή ομάδα. Εκτός απο αυτό, ο δυναμικός αλγόριθμος παρέχει το πλεονέκτημα ότι ο χρήστης (προγραμματιστής) δεν χρειάζεται να ορίσει κατάλληλα ο ίδιος τον αριθμό των παραγωγών και των καταναλωτών, αφού αυτό γίνεται δυναμικά κατα την διάρκεια της εκτέλεσης.

Για αρκετά μεγάλα όμως αρχεία εισόδου, οι δύο υλοποιήσεις μας (στατικός και δυναμικός αλγόριθμος διασωλήνωσης) έχουν πολύ κακή επίδοση και αυτό οφείλεται στον αυστηρό συγχρονισμό που υπάρχει μεταξύ των παραγωγών και των καταναλωτών για την μεταξύ τους επικοινωνία. Σε αυτή την περίπτωση, επίσης, είδαμε πώς ο δυναμικός αλγόριθμος είναι πολύ χειρότερος απο ότι ο στατικός αλγόριθμος που υλοποιήσαμε. Αυτό, εξαιτίας του ακόμα αυστηρότερου συγχρονισμού μεταξύ των νημάτων, αφού σε αυτή την υλοποίηση όλα τα νήματα μίας ομάδας έχουν πολλές κοινόχρηστες μεταβλητές τις οποίες κλειδώνουν κάθε φορά που πάνε να γράψουν σε αυτές.

Το βασικότερο συμπέρασμα στο οποίο έχουμε καταλήξει, είναι ότι για τις εφαρμογές που χρησιμοποιήσαμε, ο παραλληλισμός με διασωλήνωση δεν είναι ποτέ καλύτερος απο τον παραλληλισμό δεδομένων. Αυτό, όπως εξηγήσαμε και στο προηγούμενο κεφάλαιο, οφείλεται στον συγχρονισμό που πρέπει να υφίσταται μεταξύ των νημάτων στις υλοποιήσεις με διασωλήνωση, κάτι το οποίο αποτελεί στένωση στην επίδοση των εφαρμογών που χρησιμοποιήσαμε.

Συνεπώς, οι εφαρμογές με παρόμοια χαρακτηριστικά με αυτά των εφαρμογών που χρησιμοποιήσαμε σε αυτή την διπλωματική εργασία, είναι καλύτερα να παραλληλοποιούνται με χρήση παραλληλισμού δεδομένων και δυναμικό καταμερισμό εργασίας, αφού ο παραλληλισμός με διασωλήνωση, εκτός του ότι έχει ιδιαίτερη δυσκολία στην υλοποίησή του, δεν παράγει το ίδιο καλά αποτελέσματα που παράγονται με παραλληλισμό δεδομένων.

Τέλος, μέσα απο την εργασία αυτή, έχουμε δεί την αξία του παραλληλισμού στις εφαρμογές Βάσεων Δεδομένων, αφού και στις τρείς εφαρμογές που χρησιμοποιήθηκαν, έχει επιτευχθεί πολύ ψηλή επιτάχυνση, τόσο με διασωλήνωση (εκτός στην εφαρμογή Query-3-Hashing για τους λόγους που έχουμε εξηγήσει), τόσο και με παραλληλισμό δεδομένων.

7.2 Μελλοντική Εργασία

Μέχρι ενός βαθμού, μπορούμε να πούμε πώς είμαστε ευχαριστημένοι με τους αλγόριθμους που έχουμε υλοποιήσει και οι οποίοι κάνουν χρήση του παραλληλισμού με διασωλήνωση για την βελτίωση της επίδοσης εφαρμογών Βάσεων Δεδομένων. Επίσης, κάναμε μια αξιόλογη μελέτη μέσω της οποίας δείχνουμε πώς ο αριθμός που ορίζουμε για τους παραγωγούς και τους καταναλωτές στο μοντέλο παραγωγού-καταναλωτή μπορεί να επηρεάσει σοβαρά την επίδοση μιας εφαρμογής. Αυτό για το οποίο δεν είμαστε ευχαριστημένοι, είναι η κακή επίδοση των αλγορίθμων που υλοποιήσαμε για πολύ μεγάλο όγκο δεδομένων. Στο μέλλον λοιπόν, μπορεί να γίνει μία μελέτη η οποία να εστιάζεται στην δημιουργία ενός βέλτιστου αλγόριθμου διασωλήνωσης, μέσω του οποίου θα αποφεύγονται οι καθυστερήσεις που έχουν οι δικές μας υλοποιήσεις και που αποτελούν σοβαρές στενώσεις στην επίδοση.

Άλλη μελλοντική εργασία, θα μπορούσε να είναι η υλοποίηση άλλων έτοιμων αλγορίθμων που χρησιμοποιούν παραλληλισμό με χρήση διασωλήνωσης. Με αυτό τον τρόπο, μπορεί να γίνει σύγκριση των έτοιμων αυτών αλγορίθμων με

τις δικές μας υλοποιήσεις, ώστε να δούμε που υπερτερεί και που υστερεί η δική μας προσέγγιση.

Στην δική μας εργασία, καταλήξαμε στο συμπέρασμα ότι ο παραλληλισμός δεδομένων προσφέρει πιο ψηλή επίδοση απο ότι ο παραλληλισμός με διασωλήνωση σε εφαρμογές Βάσεων Δεδομένων. Αυτό, μπορεί να προέκυψε απο την πειραματική αξιολόγηση εφαρμογών με διαφορετικά χαρακτηριστικά, όμως ένα κοινό χαρακτηριστικό αυτών των εφαρμογών είναι το ότι έχουν μικρό αριθμό απο λειτουργίες σύνδεσης. Στο μέλλον λοιπόν, μπορεί να γίνει η σύγκριση των δύο μορφών παραλληλισμού μέσω των δικών μας υλοποιήσεων σε διαφορετικές όμως εφαρμογές με μεγάλο αριθμό απο λειτουργίες σύνδεσης. Όταν έχουμε μεγάλο αριθμό απο εργασίες, η χρήση της διασωλήνωσης καθιστά πιο εύκολη την υλοποίηση αλλά και την κατανόηση του κώδικα της εφαρμογής από μέρους του προγραμματιστή [15]. Αυτό, επειδή “σπάζουμε” τον μεγάλο κώδικα σε πιο μικρά κομμάτια κώδικα. Αυτό που μπορεί να γίνει λοιπόν, είναι να γίνει σύγκριση του παραλληλισμού με διασωλήνωση με τον παραλληλισμό δεδομένων σε τέτοιες εφαρμογές και το μέτρο σύγκρισης να μήν είναι μόνο η επιτάχυνση, αλλά και η ευκολία στην υλοποίηση και την κατανόηση του παραλληλισμού για τις συγκεκριμένες εφαρμογές.

Μια τελευταία ιδέα, είναι και η πειραματική αξιολόγηση της διασωλήνωσης στην γλώσσα C++ μέσω της βιβλιοθήκης PIPAPI [12]. Η συγκεκριμένη βιβλιοθήκη, προσφέρει στον προγραμματιστή την δυνατότητα να εκτελέσει παράλληλα μια εφαρμογή με την χρήση διασωλήνωσης. Το μόνο που έχει να κάνει ο προγραμματιστής, είναι να ορίσει ποιο μέρος της εφαρμογής θα εκτελεστεί από τους παραγωγούς και ποιο μέρος από τους καταναλωτές, καθώς επίσης και να ορίσει τις κατάλληλες κλήσεις συγκεκριμένων συναρτήσεων της βιβλιοθήκης αυτής στον κώδικά του.

Βιβλιογραφία

[1] R. Elmasri and S. B. Navathe, “Database Systems: Models, Languages, Design and Application Programming,”, 6th ed.

[2] W. Hasan, D. Florescu, P. Valduriez, “Open Issues in Parallel Query Optimization,” in *ACM SIGMOD Record,* 1996, pp. 28-33

[3] A. Gruenheid, E. Omiecinski, L. Mark, “Query Optimization Using Column Statistics in Hive,” in *Proceedings of the 15th Symposium on International Database Engineering & Applications Conference,* 2011, pp. 97-105

[4] S. Wu, F. Li, S. Mehrotra, B. C. Ooi, “Query Optimization for Massively Parallel Data Processing,” in *International Conference on Management of Data,* 2011

[5] D. DeWitt, J. Gray, “Parallel database systems: the future of high performance database systems,” in *Communications of the ACM,* 1992, pp 85-98

[6] G. Graefe, “Encapsulation of parallelism in the volcano query processing system,” in *International Conference on Management of Data,* 1990, pp. 102-111

[7] W. Hong, M. Stonebraker, “Optimization of parallel query execution plans in xprs,” in *Proceedings of the first international conference on Parallel and distributed information systems,* 1991, pp. 218-225

[8] A. Deshpande, L. Hellerstein, “Flow Algorithms for Parallel Query Optimization,” in *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering,* 2008, pp. 754-763

[9] K. Hahn, B. Reiner, “Intra-Query Parallelism for Multidimensional Array Data,”

[10] M. Shell, H. Simpson, J. Kirk, “Exploiting Scalable Many-Core Architectures for Decision Support System Workloads: Optimizing TPC-H Queries on the Intel SCC,”

[11] A. Askarunisa, P. Prameela, N. Ramraj, “DBGEN - Database (Test) GENerator - An Automated Framework for Database Application Testing,”, *International Joyrnal of Database Theory and Application, Vol.2, No. 3, September 2009*

[12] PIPAPI, Generated by Doxygen 1.8.0, Mon Mar 26 2012

[13] S. Blanas, Y. Li, J. M. Patel. “Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs,” in *International Conference on Management of Data,* 2011, pp. 37-48

[14] S. N. Mehmood, N. Haron, V. Akhtar, Y. Javed, “Implementation and Experimentation of ProducerConsumer Synchronization Problem,” in *International Journal of Computer Applications, 2011*

[15] C. Bienia, and K. Li, “Characteristics of Workloads Using the Pipeline Programming Model,” in *International Symposium on Computer Architecture,* 2012, pp. 161-171

[16] M.Kersten, S.Manegold, P.Boncz, N.Nes, “Macro- and Micro- Parallelism in a DBMS,” in *Euro-Par '01 Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing, 2001,* pp. 6-15

[17] http://daugerresearch.com/vault/SharedMemoryModel.gif

[18] http://daugerresearch.com/vault/DistributedMemoryModel.gif

[19] http://oreilly.com/catalog/oraclepp/chapter/OPP.0104.gif

[20] http://etutorials.org/shared/images/tutorials/tutorial_93/02fig02.gif

[21] http://i.msdn.microsoft.com/dynimg/IC164387.gif

[22] https://computing.llnl.gov/tutorials/parallel_comp/images/data_parallel_model.gif

[23] http://blog.springsource.com/wp-content/uploads/2011/01/amqp.jpg

[24] http://upload.wikimedia.org/wikipedia/commons/thumb/1/1c/AMD_Opteron_Six_Cores.jpg/220px-AMD_Opteron_Six_Cores.jpg