Ατομική Διπλωματική Εργασία

ΜΕΛΕΤΗ ΘΕΡΜΟΚΡΑΣΙΑΣ ΠΟΛΥΠΥΡΗΝΟΥ ΕΠΕΞΕΡΓΑΣΤΗ ΜΕ ΕΦΑΡΜΟΓΕΣ PHOENIX

Σελεάρη Φρόσω

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2010

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΜΕΛΕΤΗ ΘΕΡΜΟΚΡΑΣΙΑΣ ΠΟΛΥΠΥΡΗΝΟΥ ΕΠΕΞΕΡΓΑΣΤΗ ΜΕ ΕΦΑΡΜΟΓΕΣ PHOENIX

Σελεάρη Φρόσω

Επιβλέπων Καθηγητής

Pedro Trancoso

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

Μάιος 2010

Ευχαριστίες

Σε αυτό το σημείο θα ήθελα να ευχαριστήσω, τον επιβλέποντα καθηγητή μου κ. Pedro Trancoso για την υποστήριξη, την καθοδήγηση και τις συμβουλές που μου προσέφερε καθ' όλη τη διάρκεια αυτής της Διπλωματικής Εργασίας.

Ιδιαίτερες ευχαριστίες, θα ήθελα να δώσω στο διδακτορικό φοιτητή Παναγιώτη Πετρίδη, του οποίου τόσο η βοήθεια σε διάφορες δυσκολίες που είχα να αντιμετωπίσω, όσο και η ψυχολογική υποστήριξη ήταν καθοριστική.

Επίσης θα ήθελα, να ευχαριστήσω τους συμφοιτητές μου από την ομάδα του CASPER για τις εποικοδομητικές συζητήσεις που βοήθησαν στην εργασία μου και το ευχάριστο περιβάλλον που επικρατούσε στο εργαστήριο.

Περίληψη

Τα τελευταία χρόνια οι περισσότερες μελέτες που αφορούν την αρχιτεκτονική των υπολογιστών επικεντρώνονται στην τοποθέτηση πολλών πυρήνων στο ίδιο κύκλωμα (chip) έτσι ώστε κάποιες εργασίες να γίνονται παράλληλα. Αυτή η διαδικασία κατά την οποία θα εκτελεστούν παράλληλα οι διάφορες εργασίες ονομάζεται παράλληλη επεξεργασία και έχει ως σκοπό την εκτέλεση διαφόρων ενεργειών και την επεξεργασία περισσότερων δεδομένων σε λιγότερο χρόνο. Παρόλο που ο χρόνος εκτέλεσης διαδραματίζει έναν από τους πιο σημαντικούς ρόλους στην ανάπτυξη της παράλληλης επεξεργασίας, υπάρχει και το ζήτημα της θερμότητας το οποίο μπορεί να παίξει καθοριστικό ρόλο. Στα παράλληλα συστήματα η θερμότητα που παράγεται αποτελεί ένα πολύ σημαντικό ζήτημα. Τα ποσοστά θερμότητας που παράγονται πρέπει να είναι περιορισμένα, αλλιώς αρχίζουν να εμφανίζονται προβλήματα στο σύστημα. Ένα από τα προβλήματα είναι η υπερθέρμανση του υλικού, το οποίο μπορεί να δημιουργήσει φθορές και αστοχίες στο υλικού. Επίσης έχει παρατηρηθεί ότι η αυξημένη θερμότητα σχετίζεται με την αυξημένη κατανάλωση ισχύος που χρειάζεται το σύστημα για να μπορεί να λειτουργήσει. Τα διάφορα κυκλώματα του συστήματος ζητούν κάποια ποσότητα ισχύς/ενέργειας για να μπορέσουν να λειτουργήσουν σωστά, με αποτέλεσμα όταν ζητούν περισσότερες ποσότητες να αυξάνεται και η θερμότητα του συστήματος. Αυτό φαίνεται να είναι ένα πολύ σοβαρό πρόβλημα, αφού η αυξημένη κατανάλωση ισχύος, ουσιαστικά αποτελεί περισσότερη κατανάλωση ηλεκτρικού ρεύματος. Το ηλεκτρικό ρεύμα είναι κάτι το οποίο κοστίζει σε οικονομικό επίπεδο και χρειάζεται να μειωθεί όσο γίνεται.

Συγκεκριμένα στην εργασία μου θα μελετήσω τη θερμοκρασία που παρατηρείται σε εφαρμογές του προγραμματιστικού μοντέλου MapReduce σε πολυπύρηνο επεξεργαστή. Το προγραμματιστικό μοντέλο MapReduce είναι ευρέως διαδεδομένο και χρησιμοποιείται από μεγάλα κέντρα δεδομένων (data centers). Συγκεκριμένα εκμεταλλεύονται την ιδέα του μοντέλου αυτού, η οποία βασίζεται στην επεξεργασία και παραγωγή μεγάλου όγκου δεδομένων. Αρχικά η ιδέα αυτή εφαρμόστηκε σε cloud computing networks, πάνω στην οποία έγιναν και οι πρώτες υλοποιήσεις. Παρόλα αυτά υπάρχει η ανάγκη να προχωρήσουμε σε νέες τεχνολογίες όπως είναι οι τεχνολογίες των πολυπύρηνων επεξεργαστών. Μέχρι τώρα έχουν γίνει διάφορες έρευνες ώστε να προσαρμόσουν το μοντέλο MapReduce σε συστήματα πολυεπεξεργαστών, κάτι το οποίο έχει επιτευχθεί με την υλοποίηση του Phoenix.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο

Το προγραμματιστικό μοντέλο MapReduce [4], έχει προταθεί πολύ πρόσφατα και η αρχική ιδέα είχε γνώμονα τα κατανεμημένα υπολογιστικά συστήματα. Εμφανίστηκε ως ένα μοντέλο με πολύ καλές επιδόσεις κυρίως σε εφαρμογές που έχουν σχέση με επεξεργασία πολύ μεγάλου όγκου δεδομένων, τα οποία έχουν κυρίως μορφή κειμένου. Παρόλο που έχει προταθεί από την Google, έχουν δείξει ενδιαφέρον και άλλοι μεγάλοι οργανισμοί όπως η Yahoo! και το Facebook.

Στη συνέχεια, παρουσιάστηκε η πρόκληση να εφαρμοστεί το προγραμματιστικό μοντέλο σε συστήματα με διαφορετική δομή από την δομή των κατανεμημένων συστημάτων. Κάπως έτσι δημιουργήθηκαν οι αντίστοιχες υλοποιήσεις στις κάρτες γραφικών, στον επεξεργαστή Cell και σε συστήματα πολυεπεξεργαστών, καθιστώντας το μοντέλο ως ένα πολύ σημαντικό προγραμματιστικό μοντέλο το οποίο μπορεί να προσαρμόζεται σε διάφορες αρχιτεκτονικές.

Παρόλο το ενδιαφέρον που παρουσίασαν οι επιδόσεις του μοντέλου στις διάφορες αρχιτεκτονικές, τελευταία άρχισε να εμφανίζεται ενδιαφέρον για την κατανάλωση ενέργειας και την θερμότητα που παράγεται στα διάφορα συστήματα. Οι πρώτες

σχετικές έρευνες έγιναν στα κατανεμημένα συστήματα και σε υπολογιστές οι οποίοι σχηματίζουν συστάδα υπολογιστών (clusters)[3] .

Εξ' όσο γνωρίζουμε μέχρι τώρα δεν έχει γίνει κάποια σχετική έρευνα για τη θερμότητα που παράγεται με τη χρήση του συγκεκριμένου μοντέλου σε συστήματα πολυεπεξεργαστών. Η εργασία μου, θα επικεντρωθεί στην εκτέλεση εφαρμογών του μοντέλου MapReduce υλοποιημένες στο Phoenix[2] και θα γίνει καταγραφή των τιμών της θερμοκρασίας που παρουσιάζουν οι πυρήνες του συστήματος. Τα αποτελέσματα θα έρθουν σε σύγκριση με τις αντίστοιχες εφαρμογές υλοποιημένες με pthreads[2]. Αναμένω ότι δεν θα υπάρχουν αρκετές διαφορές στις τιμές της θερμοκρασίας που θα παρουσιάσουν οι δύο υλοποιήσεις. Κάτι αντίστοιχο συμβαίνει και στη σύγκριση που έγινε σε σχετικό άρθρο[2], για το χρόνο εκτέλεσης, δηλαδή για τις ίδιες εφαρμογές και τον ίδιο όγκο δεδομένων ο χρόνος εκτέλεσης ήταν περίπου ο ίδιος στις πλείστες περιπτώσεις.

1.2 Σκοπός

Έχοντας παρατηρήσει την έμφαση που δίνεται στο χρόνο εκτέλεσης των εφαρμογών όταν αυτές τρέχουν στις νέες τεχνολογίες επεξεργαστών (multi-core / multiprocessor), οι ερευνητές επικεντρώνονται στις μελέτες τους, στην επίδοση και αγνοούν άλλο ένα πολύ σημαντικό παράγοντα, που είναι η θερμοκρασία. Ένας παράγοντας, στον οποίο εάν δεν δώσουμε προσοχή μπορεί να προκαλέσει σημαντικά προβλήματα στο υλικό του συστήματος μας. Επίσης η αύξηση στη θερμοκρασία του συστήματος σχετίζεται με την περισσότερη παραγωγή θερμότητας, η οποία είναι αποτέλεσμα της αυξημένης ζήτησης ισχύος/ενέργειας από τα διάφορα κυκλώματα. Η αύξηση στην κατανάλωση ενέργειας, μπορεί να ερμηνευθεί είτε σαν ψηλότερο κόστος ηλεκτρικού ρεύματος, είτε σαν ένας παράγοντας που επηρεάζει την καταστροφή του περιβάλλοντος.

Έτσι, όπως έχω ήδη προαναφέρει, έχω αποφασίσει η εργασία μου να επικεντρωθεί σε σχετικές παρατηρήσεις σχετικά με τη θερμοκρασία των πυρήνων σε multi-core ή multiprocessor τεχνολογίες. Βασικός στόχος ήταν να πάρω εφαρμογές από την υλοποίηση του Phoenix [2], και με τις απαραίτητες αλλαγές να πάρω μετρήσεις σχετικά με τη θερμοκρασία που έχουν οι πυρήνες όταν τρέχουν οι διάφορες εφαρμογές.

Με αυτό τον τρόπο θα δω κατά πόσο οι νέες αυτές τεχνολογίες δίνουν σημαντικό πλεονέκτημα σε εφαρμογές που χρησιμοποιούνται από διάφορα κέντρα δεδομένων (data centers), τα οποία χρησιμοποιούν τέτοια συστήματα για να τρέξουν εφαρμογές βασισμένες στο προγραμματιστικό μοντέλο MapReduce. Με αυτά τα συμπεράσματα, οι ιδιοκτήτες τέτοιων κέντρων δεδομένων (data centers) θα μπορούν να αποφασίσουν με βάση την επίδοση που τους προσφέρει το συγκεκριμένο μοντέλο στις νέες τεχνολογίες, κατά πόσο πρέπει να αντικαταστήσουν τις παλιές.

1.3 Προγραμματιστικό μοντέλο MapReduce

Το προγραμματιστικό μοντέλο MapReduce έχει προταθεί αρχικά από την Google το 2004. Εμπνευστές αυτής της ιδέας ήταν οι Jeffrey Dean και Sanjay Ghemawat [4], οι οποίοι ήθελαν να αξιοποιήσουν τον παραλληλισμό που τους προσφέρουν συστάδες υπολογιστών (clusters). Έτσι σκοπός του συγκεκριμένου μοντέλου είναι να επεξεργάζεται τεράστιο όγκο δεδομένων και να παράγει τα κατάλληλα αποτελέσματα με τη βοήθεια ενός συνόλου συστάδων υπολογιστών (cluster).

Κύριο χαρακτηριστικό του συγκεκριμένου προγραμματιστικού μοντέλου είναι οι δύο βασικές συναρτήσεις: η map και η reduce, τις οποίες πρέπει να ορίσει ο χρήστης για τους υπολογισμούς και τον τρόπο επεξεργασίας που θα γίνεται στα διάφορα δεδομένα εισόδου.

Γενικά, το MapReduce είναι ένα προγραμματιστικό μοντέλο το οποίο εκφράζει απλούς υπολογισμούς, αποκρύπτοντας από τον προγραμματιστή της εφαρμογής λεπτομέρειες σχετικά με τον παραλληλισμό, τη διαχείριση σφαλμάτων, την κατανομή δεδομένων και το υλικό πάνω στο οποίο τρέχει κάθε εφαρμογή. Επίσης, έχει τη δυνατότητα να προσαρμόζεται εύκολα σε διαφορετικό αριθμό δεδομένων κάθε φορά. Σύμφωνα με την αρχική ιδέα το συγκεκριμένο προγραμματιστικό μοντέλο μπορεί να προσαρμόζεται από μικρές συστάδες υπολογιστών (clusters) με μερικές μόνο μηχανές, μέχρι τεράστιες συστάδες με χιλιάδες μηχανές.

Όπως έχω ήδη αναφέρει οι πρώτες υλοποιήσεις αυτού του προγραμματιστικού μοντέλου έγιναν με βάση συστάδα υπολογιστών (clusters), για να ακολουθήσουν

υλοποιήσεις σε συστήματα πολυεπεξεργαστών, στον Cell[5] και σε κάρτες γραφικών (GPUs)[1]. Για τις ανάγκες τις συγκεκριμένης διπλωματικής εργασίας, θα χρησιμοποιήσω την υλοποίηση Phoenix, στόχο είχε την εφαρμογή του MapReduce μοντέλου σε συστήματα πολυεπεξεργαστών. Το Phoenix ακολουθεί τα κύρια χαρακτηριστικά του προγραμματιστικού μοντέλου MapReduce. Αποτελείται από το "runtime system" και το "Application Programming Interface (API)". Η εκτέλεση των εφαρμογών υποστηρίζεται από το runtime system το οποίο λειτουργεί αυτόματα, αποκρύπτοντας πολλές διαδικασίες και λεπτομέρειες από το χρήστη. Το API αποτελείται από συναρτήσεις έτοιμες για να χρησιμοποιηθούν από το χρήστη και συναρτήσεις οι οποίες πρέπει ο χρήστης να ορίσει τη λειτουργία τους ανάλογα με την εφαρμογή που θέλει να φτιάξει. Στο API περιλαμβάνονται και οι δύο βασικές συναρτήσεις map και reduce.

Στη συνέχεια θα παρουσιαστούν λεπτομέρειες σχετικά με το προγραμματιστικό μοντέλο, το σκοπό για τον οποίο δημιουργήθηκε και κάποιες λεπτομέρειες για τον τρόπο λειτουργίας του. Ακολούθως θα παρουσιαστούν οι διάφορες υλοποιήσεις στις οποίες έχει εφαρμοστεί το μοντέλο, ενώ θα δοθεί περισσότερη έμφαση στην υλοποίηση Phoenix, υλοποίηση την οποία χρησιμοποιώ για να πάρω μετρήσεις της θερμοκρασίας.

1.4 Μεθοδολογία

Για την εκπόνηση της συγκεκριμένης διπλωματικής εργασίας χρειάστηκαν ουσιαστικά 4 βασικά βήματα.

Σαν πρώτο βήμα, είχα να μελετήσω σε λεπτομέρεια το προγραμματιστικό μοντέλο MapReduce, να δω ποιός είναι ο σκοπός του και τις διάφορες υλοποιήσεις που έγιναν θέτοντας το σαν βάση.

Σαν δεύτερο βήμα, χρειάστηκε να μελετήσω τα διάφορα εργαλεία, ώστε να γνωρίζω τη χρήση τους και τον τρόπο με τον οποίο λειτουργούν. Για το συγκεκριμένο προγραμματιστικό μοντέλο είχα επιλέξει την υλοποίηση του Phoenix. Η συγκεκριμένη υλοποίηση έχει επιλεγεί επειδή έχει σχεδιαστεί για εκτέλεση σε συστήματα πολυεπεξεργαστών. Επίσης αυτή η υλοποίηση τρέχει σε περιβάλλον SOLARIS ή

LINUX. Το δεύτερο εργαλείο το οποίο είχα επιλέξει για αυτή την εργασία ήταν το Lm_sensors[9]. Είναι ένα εργαλείο με το οποίο μπορώ να πάρω μετρήσεις για τη θερμοκρασία των πυρήνων του συστήματος. Είχα καταλήξει σε αυτό το εργαλείο, αφού είχα μελετήσει και δοκιμάσει αρκετά εργαλεία μέτρησης της θερμοκρασίας. Το εργαλείο αυτό έχει δημιουργηθεί για να τρέχει σε περιβάλλον LINUX και οι κριτικές οι οποίες έχουν ειπωθεί, αναφέρουν ότι είναι ένα εργαλείο με αρκετή ακρίβεια στις μετρήσεις που δίνει. Επίσης ένα σημαντικό πλεονέκτημα του συγκεκριμένου εργαλείου ήταν η δυνατότητα να ενσωματωθεί μέσα στων κώδικα των εφαρμογών, σε αντίθεση με άλλα αντίστοιχα εργαλεία.

Κατά το τρίτο βήμα, και μετά την πρώτη επαφή με τα διάφορα εργαλεία, άρχισα να μελετώ των κώδικα των εφαρμογών που βρίσκονταν υλοποιημένες στο πακέτο της υλοποίησης του Phoenix. Σε αυτό το σημείο αρχικά έπρεπε να ενσωματώσω το εργαλείο Lm_sensors στις εφαρμογές της υλοποίησης του Phoenix, για να μπορώ να παίρνω μετρήσεις στη θερμοκρασία κατά τη διάρκεια της εκτέλεσης των εφαρμογών. Έπειτα χρειάστηκε να προσαρμόσω των κώδικα των εφαρμογών, ώστε να μπορώ να καθορίζω εγώ κάποιες παραμέτρους κατά την εκτέλεση. Ουσιαστικά οι αλλαγές γίνονταν στο API μέσω των εφαρμογών, ενώ κάποια στιγμή χρειάστηκε να γίνουν αλλαγές και στο κώδικα του runtime system.

Τέλος, αφού τελείωσε το τρίτο βήμα που ήταν και το πιο χρονοβόρο βήμα, και πήρα τις απαραίτητες μετρήσεις, είχαμε ορίσει σαν τέταρτο βήμα την υλοποίησης δικών μας εφαρμογών στην υλοποίηση του Phoenix και τη βοήθεια του API που προσφέρει. Οι εφαρμογές που άρχισα να υλοποιώ ανήκουν στις εφαρμογές του TPC-H [10].

1.5 Οργάνωση

Το Κεφάλαιο 1 ήταν ένα εισαγωγικό κεφάλαιο, στο οποίο γίνεται αναφορά για το τι θα ακολουθήσει. Στο Κεφάλαιο 2 θα επικεντρωθώ στη σχετική δουλειά που έγινε για το προγραμματιστικό μοντέλο MapReduce και στα συστήματα πολυεπεξεργαστών. Στο Κεφάλαιο 3 θα δώσω κάποιες περισσότερες λεπτομέρειες για την υλοποίηση του Phoenix, υλοποίηση που έγινε για το προγραμματιστικό μοντέλο mapReduce σε συστήματα πολυεπεξεργαστών. Το Κεφάλαιο 4 είναι το κεφάλαιο στο οποίο θα μιλήσω

για το πώς η θερμότητα επηρεάζει τα συστήματα υπολογιστών και τον τρόπο με τον οποίο θα πάρω τις διάφορες μετρήσεις της θερμοκρασίας από τους πυρήνες όταν τρέχουν εφαρμογές του Phoenix. Το Κεφάλαιο 5 επικεντρώνεται στις εφαρμογές του TPC-H και τις σχετικές υλοποιήσεις που έκανα. Το Κεφάλαιο 6 παρουσιάζει τα αποτελέσματα από τις μετρήσεις στη θερμοκρασία όταν έτρεξα τις εφαρμογές του Phoenix και τα αποτελέσματα του χρόνο εκτέλεσης από τις υλοποιήσεις των εφαρμογών του TPC-H. Τέλος στο Κεφάλαιο 7, θα παρουσιάσω τα συμπεράσματα που εξάγονται από τις διάφορες μετρήσεις και κάποιες ιδέες οι οποίες μπορούν να γίνουν στο μέλλον.

Κεφάλαιο 2

Σχετική Δουλειά

2.1 Παράλληλα Συστήματα

Στην έννοια του παραλληλισμού οδήγησε η ανάγκη για καλύτερη επίδοση από τα διάφορα υπολογιστικά συστήματα. Στην ανάπτυξη του βοήθησε το γεγονός ότι αρκετές εφαρμογές μπορούν να εκτελέσουν κάποιο κομμάτι τους σειριακά και κάποιο άλλο παράλληλα. Έτσι άρχισαν οι προσπάθειες για εκμετάλλευση του παραλληλισμού με το σχεδιασμό των παράλληλων εκδοχών διάφορων εφαρμογών αλλά και με το σχεδιασμό νέων τεχνολογιών, που στόχο έχουν την καλύτερη επίδοση.

Αρχικά στις νέες τεχνολογίες που σχεδιάζονταν παρατηρήθηκε συνεχής αύξηση της συχνότητας του επεξεργαστή, μονάδα η οποία εκφράζει τον αριθμό των εντολών που εκτελούνται σε κάθε κύκλο μηχανής. Παράλληλα υπήρχε αύξηση του αριθμού των τρανζίστορς που χρησιμοποιούνται για την κατασκευή του κυκλώματος του επεξεργαστή, κάτι το οποίο αυξάνει την πολυπλοκότητα του. Ο ρυθμός αύξησης του αριθμού των τρανζίστορς περιγράφεται από το νόμο του Moore, ο οποίος προβλέπει ότι κάθε 18 μήνες διπλασιάζεται ο αριθμός των τρανζίστορς στα κυκλώματα κάποιου

επεξεργαστή. Κάτι τέτοιο ήταν εφικτό αφού υπήρχε ταυτόχρονα μείωση στο μέγεθος των τρανζίσοτρς χωρίς να γίνεται κάποια αύξηση στο μέγεθος του κυκλώματος (chip).

Η συνεχής αύξηση της συχνότητας του επεξεργαστή, δηλαδή η εκτέλεση περισσότερων εντολών σε κάθε κύκλο μηχανής και η αύξηση της πολυπλοκότητας, σαν αποτέλεσμα της μείωσης του μεγέθους των τρανζίστορς σε συνδυασμό με την αύξηση του αριθμού των τρανζίστορς σε ένα επεξεργαστή, οδήγησαν σε ένα νέο πρόβλημα, την αύξηση της κατανάλωσης της ενέργειας. Η ενέργεια που παρατηρείται μετατρέπεται σε θερμότητα, η οποία μπορεί να προκαλέσει ζημιά στους υπολογισμούς που γίνονται λόγω φθοράς ή καταστροφής στο υλικό της μηχανής.

Για να αποφευχθεί αυτό το πρόβλημα, υπήρξε καινοτομία στην αρχιτεκτονική των επεξεργαστών, αφού έκαναν την εμφάνιση τους οι γνωστοί σε εμάς πολυπύρηνοι επεξεργαστές. Σε αυτή την αρχιτεκτονική είχαν ενσωματώσει πολλούς πυρήνες σε ένα κύκλωμα (chip), ώστε να μπορέσουν να βελτιώσουν την επίδοση των υπολογιστικών μηχανών. Οι πυρήνες που είχαν ενσωματωθεί αρχικά στο κύκλωμα (chip), αποτελούνταν από απλά κυκλώματα και χρειάζονται λιγότερα τρανζίστορς για τη σύνθεση τους.

Εκτός από την εμφάνιση των πολυπύρηνων επεξεργαστών, έκαναν την εμφάνιση τους και άλλες αρχιτεκτονικές οι οποίες χρησιμοποιούν με κάποιο τρόπο τον παραλληλισμό. Πιο σημαντικές αρχιτεκτονικές, παρουσιάζονται οι αρχιτεκτονικές που χρησιμοποιούνται στις κάρτες γραφικών (GPUs) και στον επεξεργαστή CELL.

Οι κάρτες γραφικών χρησιμοποιούνται κυρίως για εφαρμογές γραφικών. Ένα GPU αποτελεί ένα τμήμα του υπολογιστή το οποίο λαμβάνει δεδομένα από την Κεντρική Μονάδα Επεξεργασίας (CPU), τα οποία θα μετατρέψει σε εικόνα. Αποτελούνται κυρίως από πάρα πολλούς επεξεργαστές, ο αριθμός τους ανέρχεται πολλές φορές σε μερικές εκατοντάδες. Οι επεξεργαστές οι οποίοι συνθέτουν μια κάρτα γραφικών βοηθούν στην παράλληλη εκτέλεση πολύπλοκων μαθηματικών και γεωμετρικών υπολογισμών που είναι απαραίτητοι για την απόδοση των γραφικών.

Ο Cell επεξεργαστής, αποτελείται από ένα κεντρικό επεξεργαστή (PPE – Power Processing Element) και οκτώ βοηθητικούς επεξεργαστές (SPEs – Synergistic Processing Elements). Οι μονάδες επεξεργασίας, η μνήμη και οι συσκευές Εισόδου/Εξόδου επικοινωνούν μεταξύ τους μέσω κάποιου είδους coherent bus, το οποίο ονομάζεται EIB – Element Interconnect Bus. Ο PPE, χρησιμοποιείται κυρίως για να τρέχει το λειτουργικό σύστημα, ενώ αναλαμβάνει και το ρόλο του συντονιστή για τους SPEs. Οι SPEs από την πλευρά τους αναλαμβάνουν να εκτελέσουν διάφορους υπολογισμούς στην τοπική τους μνήμη, τους οποίους αναθέτει ο συντονιστής, δηλαδή ο PPE. Ο cell αποτελεί πολύ σημαντικό κομμάτι για το Playstation 3 και τις διάφορες εφαρμογές του PS3.

Τα πιο πάνω συστήματα πολυεπεξεργαστών μπορούν να χωριστούν σε δύο κατηγορίες, στην κατηγορία των ομογενών συστημάτων και στην κατηγορία των ετερογενών συστημάτων. Οι πολυπύρηνοι επεξεργαστές και οι GPUs είναι συστήματα με ομοιογένεια, αφού αποτελούνται από πανομοιότυπους επεξεργαστές και πυρήνες. Σε αντίθεση με τον Cell, ο οποίος ανήκει στην κατηγορία των ετερογενών συστημάτων, αφού την αρχιτεκτονική του συνθέτουν δύο διαφορετικά είδη επεξεργαστών (PPE, SPEs). Επίσης υπάρχει η δυνατότητα να συνθέσουμε διάφορες αρχιτεκτονικές, να τις βάλουμε να λειτουργούν σε συνεργασία μεταξύ τους και έτσι να δημιουργήσουμε και άλλα είδη ετερογενών υπολογιστικών συστημάτων.

2.2 Μοντέλα Παράλληλου Προγραμματισμού

Η ύπαρξη περισσότερων από μιας υπολογιστικής μονάδας σε κάποιο σύστημα περιπλέκει την μοντελοποίηση και το προγραμματισμό σε κάποιο υπολογιστικό σύστημα. Κάτι τέτοιο οδηγεί στη δημιουργία πολλών διαφορετικών προγραμματιστικών μοντέλων ανάλογα με την αρχιτεκτονική κάθε συστήματος.

Αρχικά ο Flynn προσπάθησε να τοποθετήσει σε κατηγορίες την αρχιτεκτονική των παράλληλων υπολογιστών. Κατά κύριο λόγο είχε στηριχτεί στον αριθμό των εντολών που εκτελούνται ταυτόχρονα και στη ροή των δεδομένων του συστήματος. Οι 4 κατηγορίες τις οποίες διατύπωσε είναι οι εξής:

- SISD: Single Instruction stream, Single Data stream (σειριακό)

- MISD: Multiple Instruction stream, Single Data stream
- SIMD: Single Instruction stream, Multiple Data stream
- MIMD: Multiple Instruction stream, Multiple Data stream

Στη συνέχεια είχαν δημιουργηθεί διάφορα προγραμματιστικά μοντέλα ειδικά για κάθε ξεχωριστή αρχιτεκτονική. Αναφορικά στις κάρτες γραφικών (GPUs) το πιο διαδεδομένο προγραμματιστικό μοντέλο είναι το προγραμματιστικό μοντέλο CUDA, το οποίο προωθήθηκε από την NVIDIA. Συγκεκριμένα η CUDA, παρέχει μια υπολογιστική μηχανή η οποία επιτρέπει σε κάποιο σχεδιαστή λογισμικού (software developer) με τη βοήθεια διαφόρων γλωσσών προγραμματισμού να υλοποιήσεις εφαρμογές που να τρέχουν στις κάρτες γραφικών.

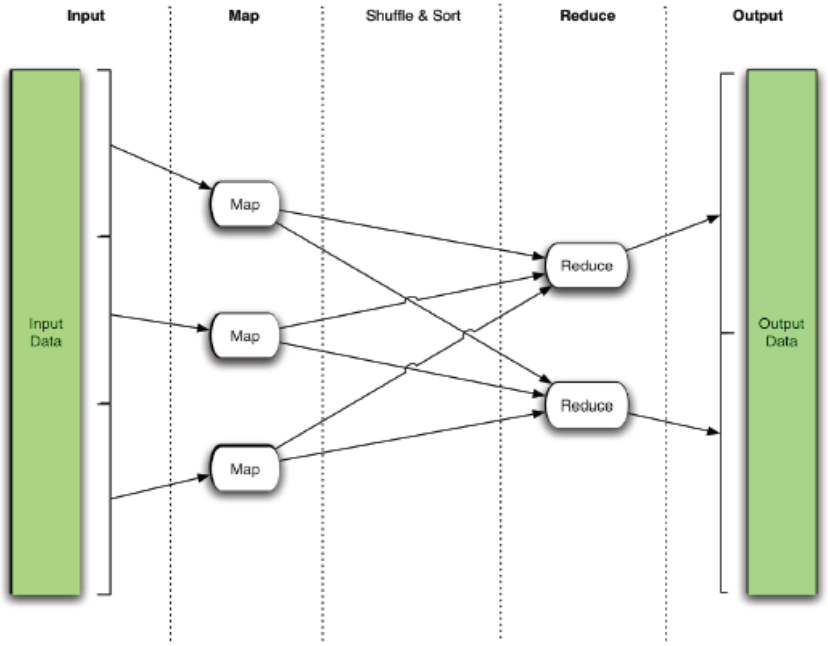
Όσον αφορά τις μηχανές οι οποίες χρησιμοποιούν πολυπύρηνους επεξεργαστές, δημιουργήθηκαν αρκετά προγραμματιστικά μοντέλα τα οποία βοηθούν στην παράλληλη εκτέλεση διαφόρων εφαρμογών. Τα πιο πολυχρησιμοποιημένα Application Program Interface(API) σε multiprocessor αρχιτεκτονικές, είναι τα Pthreads και η OpenMp.

Όταν έχουμε υπολογιστές οι οποίοι δημιουργούν μια συστάδα (cluster) ή στην περίπτωση των υπερυπολογιστών (supercomputers), η επικοινωνία ανάμεσα στις μονάδες επεξεργασίας γίνεται με την αποστολή μηνυμάτων. Για αυτή τη περίπτωση αρχιτεκτονικής, δημιουργήθηκε κάποιο API, που ονομάστηκε MPI (Message Passing Interface) και εγγυάται την παράλληλη εκτέλεση κάποιας εφαρμογής αλλά την ίδια στιγμή η επικοινωνία συνεχίζει να γίνεται με την ανταλλαγή μηνυμάτων.

2.3 Προγραμματιστικό Μοντέλο MapReduce

Η ιδέα που έδωσε το έναυσμα για να αρχίσουν μελέτες οι οποίες κατέληξαν στο MapReduce προγραμματιστικό μοντέλο, γεννήθηκε από την ανάγκη για επεξεργασία και παραγωγή τεράστιου όγκου δεδομένων. Προτάθηκε για πρώτη φορά το 2004 από την Google [4], και σκοπός ήταν η επεξεργασία τεράστιου όγκου δεδομένων με τη βοήθεια συστάδων υπολογιστών (clusters).

Για τις ανάγκες του προγραμματιστικού μοντέλου, υπάρχουν δύο βασικές συναρτήσεις, η συνάρτηση map και η συνάρτηση reduce, των οποίων οι λεπτομέρειες των εργασιών που εκτελούνται ορίζονται από το χρήστη ανάλογα με την εφαρμογή την οποία επιθυμεί. Μια MapReduce εργασία θα σπάσει τα δεδομένα εισόδου σε ανεξάρτητα κομμάτια, τα οποία θα δοθούν σε μια map εργασία το κάθε ένα και θα τύχουν επεξεργασίας παράλληλα. Κατά τη συνάρτηση map εκτελούνται κάποιες διαδικασίες με τις οποίες τα δεδομένα εισόδου, παράγουν ένα ζεύγος (key/value). Αφού τελειώσουν όλες οι map εργασίες, τα ζεύγη που θα παραχθούν, δηλαδή η έξοδος που θα δώσει κάθε map εργασία, θα δημιουργηθούν κάποια σύνολα με ενδιάμεσα ζεύγη (key/value) τα οποία θα έχουν ίδιο κλειδί, ουσιαστικά γίνεται ένα είδος ενδιάμεσης ταξινόμησης. Τα σύνολα με τα ενδιάμεσα ζεύγη στη συνέχεια θα δοθούν σαν είσοδος στις reduce εργασίες, οι οποίες και πάλι εκτελούνται παράλληλα. Η συνάρτηση reduce, έχει την ευθύνη ανάλογα με την εφαρμογή, να συγχωνεύσει τις τιμές (values) από τα ενδιάμεσα ζεύγη και να παράγει μια ενιαία τιμή για τα ζεύγη τα οποία έχουν το ίδιο κλειδί (key). Η σειρά που ακολουθείται κατά την εκτέλεση κάποιας MapReduce εφαρμογής φαίνεται στο Σχήμα 2.3.α.



Σχήμα 2.3 Βήματα εκτέλεσης MapReduce Εφαρμογής

Είναι πολύ σημαντικό ότι αυτό το προγραμματιστικό μοντέλο εκτελεί παράλληλα αρκετές εργασίες. Κάποιος προγραμματιστής όμως που θα επιλέξει να γράψει εφαρμογές στο συγκεκριμένο μοντέλο, δεν χρειάζεται να ανησυχεί για λεπτομέρειες σχετικά με τον παραλληλισμό των εργασιών που θα εκτελεστούν, αφού ορίζεται αυτόματα από το μοντέλο. Το runtime system είναι υπεύθυνο να μοιράζει τα δεδομένα, να χρονοδρομολογεί την εκτέλεση της εφαρμογής στο σύνολο των μηχανών. Επίσης χειρίζεται κάποια σφάλματα μηχανής, αναγκάζοντας κάποιες εργασίες να εκτελεστούν ξανά, και τέλος έχει την ευθύνη να διαχειρίζεται τον τρόπο με τον οποίο θα επικοινωνούν οι μηχανές μεταξύ τους, όπου είναι απαραίτητο. Για το μόνο παράλληλο θέμα που ανησυχεί το χρήστη κατά την υλοποίηση κάποιας εφαρμογής είναι πιθανές εξαρτήσεις ανάμεσα στα δεδομένα όταν θα σπάσουν σε κομμάτια.

Στη συνέχεια θα εξηγήσω ένα μικρό παράδειγμα, της εφαρμογής Word Count η οποία εφαρμόστηκε στο MapReduce. Η εφαρμογή Word Count αναλαμβάνει ένα αρχείο δεδομένων σε μορφή κειμένου και είναι υποχρεωμένη να παρουσιάσει τη συχνότητα εμφάνισης κάθε ξεχωριστής λέξης στο αρχείο, δηλαδή θα μετρήσει κάθε λέξη πόσες φορές έχει εμφανιστεί μέσα στο αρχείο.

Αρχικά το αρχείο θα σπάσει σε διάφορα μέρη. Κάθε κομμάτι του αρχείου θα δοθεί σε μια υπολογιστική μονάδα για να εκτελεστεί η συνάρτηση map. Στην προκειμένη περίπτωση θα παίρνει μια μια τις λέξεις από το κομμάτι του αρχείου που έχει αναλάβει, θα θέτει τη λέξη σαν το κλειδί του ζεύγους που θα παράγει και θα του δίνει τη τιμή 1. Με αυτό το τρόπο δηλώνει ότι η λέξη-κλειδί βρέθηκε 1 φορά. Αφού εξαντληθούν όλα τα κομμάτια του αρχείου και τελειώσουν ουσιαστικά όλες οι map εργασίες, τα ζεύγη θα ταξινομηθούν με βάση το κλειδί-λέξη και θα τοποθετηθούν σε ομάδες. Κάθε ομάδα θα περιέχει ζεύγη με το ίδιο κλειδί, στην περίπτωση μας θα περιέχει ζεύγη με την ίδια λέξη. Κάθε ομάδα θα δοθεί και πάλι σε μια υπολογιστική μονάδα για να εκτελεστεί η συνάρτηση reduce. Εδώ, η εργασία reduce θα προσθέσει την τιμή κάθε ζεύγους. Με το τέλος κάθε εργασίας θα έχουμε το συνολικό αριθμό εμφάνισης της λέξης που ανέλαβε έμμεσα μέσω της ομάδας που της δόθηκε.

Πιο κάτω παρουσιάζεται η εκτέλεση ενός παραδείγματος και στο παράρτημα Α.1 παρουσιάζεται ο ψευδοκώδικας [4] την εφαρμογής Word Count σε MapReduce.

Παράδειγμα

Αρχείο εισόδου

Καινούργιους τόπους δεν θα βρεις,
δεν θά βρεις άλλες θάλασσες.
Η πόλις θα σε ακολουθεί.

**Το αρχείο εισόδου αποτελεί στίχους από το ποίημα του Κ. Καβάφη Η Πόλις*

Το αρχείο εισόδου έσπασε σε μερικά κομμάτια και για κάθε κομμάτι εκτελέστηκε η συνάρτηση map. Από τις συναρτήσεις map έχουν παραχθεί τα πιο κάτω ζεύγη (κλειδί / τιμή).

Αποτέλεσμα Map Stage

Ζεύγος 1 («Καινούργιους», 1)
Ζεύγος 2 («τόπους», 1)
Ζεύγος 3 («δεν», 1)
Ζεύγος 4 («θα», 1)
Ζεύγος 5 («βρεις», 1)
Ζεύγος 6 («δεν», 1)
Ζεύγος 7 («θα», 1)
Ζεύγος 8 («βρεις», 1)
Ζεύγος 9 («άλλες», 1)
Ζεύγος 10 («θάλασσες», 1)
Ζεύγος 11 («Η», 1)
Ζεύγος 12 («πόλις», 1)
Ζεύγος 13 («θα», 1)
Ζεύγος 14 («σε», 1)
Ζεύγος 15 («ακολουθεί», 1)

Τα ζεύγη ταξινομούνται και τοποθετούνται σε ομάδες. Κάθε ομάδα αποτελείται από ζεύγη που έχουν σαν κλειδί την ίδια λέξη. Οι ομάδες δίνονται στις reduce εργασίες για εκτέλεση. Με το τέλος όλων των reduce εργασιών δίνεται το τελικό αποτέλεσμα που έχει ως εξής :

Αποτελέσματα Reduce Stage

Ομάδα 1 («Καινούργιους», 1)

Ομάδα 2 («τόπους», 1)

Ομάδα 3 («δεν», **2**)

Ομάδα 4 («θα», **3**)

Ομάδα 5 («βρεις», **2**)

Ομάδα 9 («άλλες», 1)

Ομάδα 10 («θάλασσες», 1)

Ομάδα 11 («Η», 1)

Ομάδα 12 («πόλις», 1)

Ομάδα 14 («σε», 1)

Ομάδα 15 («ακολουθεί», 1)

Εδώ τελειώνει η εκτέλεση του συγκεκριμένου παραδείγματος και το αποτέλεσμα δίνεται στο χρήστη.

Γενικά, το MapReduce είναι ένα προγραμματιστικό μοντέλο το οποίο εκφράζει απλούς υπολογισμούς, αποκρύπτοντας από τον προγραμματιστή της εφαρμογής λεπτομέρειες σχετικά με τον παραλληλισμό, τη διαχείριση σφαλμάτων και την κατανομή δεδομένων. Επίσης, έχει τη δυνατότητα να προσαρμόζεται εύκολα σε διαφορετικό αριθμό δεδομένων κάθε φορά. Από μικρές συστάδες υπολογιστών (clusters) με μερικές μόνο μηχανές, μέχρι τεράστιες συστάδες με χιλιάδες μηχανές

2.4 Υλοποιήσεις MapReduce

Σε αυτό το σημείο θα παρουσιάσω τις υλοποιήσεις που έχουν γίνει με βάση το προγραμματιστικό μοντέλο MapReduce.

2.4.1 Hadoop MapReduce

Το Hadoop είναι μια πολύ σημαντική υλοποίηση του προγραμματιστικού μοντέλου MapReduce, η οποία προτάθηκε από την Yahoo! [8]. Η υλοποίηση αυτή δίνει τη δυνατότητα στον προγραμματιστή να γράφει εύκολα εφαρμογές, οι οποίες

επεξεργάζονται τεράστιο όγκο δεδομένων παράλληλα, με τη βοήθεια συστάδων υπολογιστών (clusters), τα οποία αποτελούνται από πολλούς ηλεκτρονικούς υπολογιστές που υποστηρίζουν την πλατφόρμα Java, χωρίς να είναι απαραίτητο η εφαρμογές να γράφονται σε Java, αφού υπάρχει και οι δυνατότητα να γράφονται και σε άλλες γλώσσες προγραμματισμού όπως είναι η C++.

Σχεδιάστηκε με βάση την αρχική ιδέα του προγραμματιστικού αυτού μοντέλου, δηλαδή για να τρέχει σε υλικό οικιακών συσκευών, οι οποίες θα επικοινωνούν μεταξύ τους όταν σχηματίζουν συστάδα υπολογιστών (σχηματίζουν cluster). Είναι μια υλοποίηση πολύ ευαίσθητη στο να ανιχνεύει σφάλματα και μπορεί εύκολα να τα χειριστεί, ενώ σχεδιάστηκε για χαμηλού κόστους υλικό. Επίσης παρέχει πρόσβαση στα δεδομένα των εφαρμογών με υψηλό throughput, ενώ προσαρμόζεται εύκολα σε εφαρμογές με τεράστιο όγκο δεδομένων.

Για τη λειτουργία του Hadoop είναι σημαντικό τα δεδομένα εισόδου αλλά και τα αποτελέσματα να αποθηκεύονται στο HDFS (Hadoop’s Distributed File System), ένα κατανεμημένο σύστημα αποθήκευσης μεγάλου όγκου δεδομένων σε πολλές μηχανές που ανήκουν σε μια συστάδα υπολογιστών (cluster).

Το HDFS χρησιμοποιεί αρχιτεκτονική master/slave. Και συγκεκριμένα ένα HDFS cluster αποτελείται από ένα Namenode κόμβο και μερικούς Datanodes κόμβους. Ο Namenode κόμβος είναι ο master κόμβος, υπεύθυνος για τη διαχείριση του namespace του σύστημα αρχείων (file system) και τη ρύθμιση της πρόσβασης στα διάφορα αρχεία από τους πελάτες (clients). Οι Datanodes είναι συνήθως ίσοι σε αριθμό με τους κόμβους του δικτύου που θα εκτελέσουν τις map/reduce εργασίες. Είναι υπεύθυνοι να διαχειριστούν τον αποθηκευτικό τους χώρο, αφού αποθηκεύουν κομμάτια (blocks) δεδομένων τα οποία εξυπηρετούν διάφορα reads/writes υπακούοντας αιτήματα των πελατών του συστήματος αρχείων (file system’s clients). Τέλος οι Datanodes είναι υπεύθυνοι να υπακούν σε κάποιες αιτήσεις του Namenode για block creation, deletion, και replication, καθώς είναι και ο κόμβος ο οποίος χειρίζεται το mapping των διαφόρων blocks που ανήκουν σε κάθε ξεχωριστό DataNode.

Για να ολοκληρωθεί το συγκεκριμένο Map/Reduce framework χρειάζονται να υπάρχει ένας master Jobtracker, ενώ σε κάθε κόμβο του cluster πρέπει να υπάρχει ένας slave Tasktracker. Ο Jobtracker, ο οποίος έχει το ρόλο του master, είναι υπεύθυνος να ρυθμίζει την παράλληλη εκτέλεση των διαφόρων map/reduce εργασιών. Χρονοδρομολογεί τις διάφορες εργασίες (map/reduce) στους slaves, δηλαδή αναθέτει εργασίες στους slaves, ενώ παρακολουθεί την εκτέλεση τους και αν υπάρξει κάποια αποτυχία σε κάποιο κόμβο θα αναλάβει να αναθέσει ξανά την εργασία σε κάποιο κόμβο για να την εκτελέσει ξανά.

2.4.2 MapReduce for the Cell B.E Architecture

Η συγκεκριμένη εργασία είχε επικεντρωθεί στον σχεδιασμό και την υλοποίηση του MapReduce προγραμματιστικού μοντέλου, πάνω στην αρχιτεκτονική του Cell [5]. Στόχος ουσιαστικά ήταν να πάρει τα χαρακτηριστικά του συγκεκριμένου προγραμματιστικού μοντέλου και να τα εφαρμόσει στην αρχιτεκτονική του επεξεργαστή Cell. Το runtime system της συγκεκριμένης υλοποίησης αυτόματα σπάζει τα δεδομένα εισόδου, δρομολογεί τις διάφορες εργασίες των εφαρμογών στους πυρήνες, διαχειρίζεται μεταφορές δεδομένων που πρέπει να γίνουν ανάμεσα στην κοινόχρηστη (global) μνήμη και τη μνήμη των SPEs και τέλος χειρίζεται την επικοινωνία και τον συγχρονισμό ανάμεσα στους πυρήνες.

Και ενώ ο επεξεργαστής Cell απαιτεί από τον προγραμματιστή να έχει τον απόλυτο έλεγχο στο κώδικα του, δημιουργείται η συγκεκριμένη υλοποίηση η οποία λύνει τα χέρια στους διάφορους χρήστες. Τώρα κάποιος χρήστης που θέλει να δημιουργήσει μια εφαρμογή στη συγκεκριμένη πλατφόρμα και με το συγκεκριμένο προγραμματιστικό μοντέλο, θα χρειαστεί να υλοποιήσει μόνο τις map και reduce συναρτήσεις ανάλογα με την εφαρμογή του. Σύμφωνα με το runtime system, οι map και οι reduce εργασίες θα εκτελεστούν μόνο από τους SPEs και δεν θα αναμειχθεί καθόλου στην εκτέλεση τους ο PPE.

Η υλοποίηση βασίζεται σε 5 βασικά στάδια, τα οποία βοηθούν στην εκτέλεση μιας εφαρμογής, τα στάδια map, partition, quick-sort, merge-sort και reduce.

Stage 1 - Map: Κατά το στάδιο map, οι SPEs εκτελούν τη συνάρτηση map, όπως την ορίζει ο χρήστης πάνω στα δεδομένα εισόδου και παράγει ένα σύνολο κλειδιών και ένα σύνολο τιμών, ενώ τα κλειδιά έχουν ένα δείκτη ο οποίο δείχνει στην αντίστοιχη τιμή του. Το runtime system τρέχει στον PPE, και είναι υπεύθυνο να αρχικοποιήσει την εκτέλεση αυτής της φάσης και να διαχειρίζεται την κίνηση των δεδομένων ώστε να τροφοδοτεί την τοπική μνήμη των SPEs. Τα δεδομένα εισόδου διαιρούνται σε 8 κομμάτια ιδίου μεγέθους, ένα κομμάτι για κάθε SPE. Επίσης δεσμεύονται 8 buffers (ένα για κάθε SPE) στη κοινόχρηστη μνήμη όπου θα τοποθετηθούν τα δεδομένα εξόδου. Στη συνέχεια σε κάθε SPE θα δοθεί ένας δείκτης στο buffer στο οποίο θα βρίσκονται τα δεδομένα εισόδου και ένας δείκτης στο buffer για τα δεδομένα εξόδου. Κατά την παραγωγή των διαφόρων ζευγών, τα αποτελέσματα αποθηκεύονται στο τοπικό buffer κάθε SPE και όταν γεμίσει μεταφέρονται στο αντίστοιχο buffer στη κοινόχρηστη μνήμη.

Stage 2 – Partition: Στο στάδιο αυτό θα τοποθετηθούν τα ζεύγη που έχουν δημιουργηθεί σε ομάδες. Κάθε ομάδα θα περιέχει ζεύγη τα οποία έχουν το ίδιο κλειδί. Καθώς τα αποτελέσματα των SPEs στο map stage έχουν γραφτεί στη κοινόχρηστη μνήμη, θα αναλάβει ο PPE να κοιτάζει κάθε κλειδί και να το τοποθετήσει στη ανάλογη ομάδα. Ουσιαστικά ο PPE θα αναλάβει να εκτελέσει κάποιου είδους hashing.

Stage 3 – Quick-sort: Εδώ θα δοθούν τα μέχρι τώρα αποτελέσματα όπως είναι τοποθετημένα σε ομάδες, από την κοινόχρηστη μνήμη στους SPEs, θα ταξινομηθούν και μετά θα επιστραφούν ξανά στην κοινόχρηστη μνήμη.

Stage 4 – Merge Sort: Λόγω του μεγάλου όγκου δεδομένων και της μικρής σε μέγεθος τοπική μνήμη των SPEs, υπάρχει η πιθανότητα στο προηγούμενο στάδιο να μην γίνει σωστή ταξινόμηση. Έτσι δημιουργήθηκε αυτό το στάδιο, κατά το οποίο θα δοθούν στους SPEs τα sorted buffers στους οποίους θα εκτελεστεί το merge-sorting.

Stage 5 – Reduce: Κατά το reduce στάδιο, οι SPEs εκτελούν τη συνάρτηση reduce όπως την έχει ορίσει ο χρήστης (όπως και τη map συνάρτηση), για όλες τις τιμές για κάθε κλειδί. Ο PPE δίνει πληροφορίες σχετικά με τις ταξινομημένες ομάδες και ένα

δείκτη στο output buffer που δεσμεύεται για κάθε SPE, και θα γραφτεί το τελικό αποτέλεσμα τη συγχώνευση των τιμών.

Οι εφαρμογές που υλοποιήθηκαν με βάση το MapReduce μοντέλο έδειξαν ότι μπορούν να προσαρμόζονται σε διαφορετικό αριθμό SPEs είτε αυτό σημαίνει αύξηση τους είτε μείωση τους, και να συνεχίζουν να έχουν καλή επίδοση. Επίσης, ο επεξεργαστής CELL έδειξε υψηλό βαθμό επίδοσης σε εφαρμογές με πάρα πολλούς υπολογισμούς σε αντίθεση με την αδυναμία που έδειξε σε εφαρμογές με λιγότερους υπολογισμούς.

2.4.3 Mars: A MapReduce Framework on Graphics Processors

Το Mars είναι μια υλοποίηση του προγραμματιστικού μοντέλου MapReduce, πάνω σε κάρτες γραφικών (GPUs) [1]. Η ιδέα άρχισε να δημιουργείται από το γεγονός ότι οι GPUs σε σύγκριση με τους συνηθισμένους επεξεργαστές (CPUs) έχουν το πλεονέκτημα μεγαλύτερης υπολογιστικής δύναμης και μεγαλύτερου bottleneck. Βέβαια υπήρχε η πρόκληση της δυσκολίας στον προγραμματισμό μιας τέτοιας υλοποίησης αφού τα υπάρχοντα προγραμματιστικά interfaces είχαν σχέση περισσότερο με εφαρμογές γραφικών.

Αυτή η υλοποίηση στηριζόταν στη βασική ιδέα του προγραμματιστικού μοντέλου με τις δύο συναρτήσεις map και reduce, ενώ βασικός στόχος ήταν να μπορέσει να εκμεταλλευτεί τον παραλληλισμό με τη δημιουργία αρκετών threads τα οποία θα τρέχουν στους επεξεργαστές του GPU. Έτσι όπως και στις άλλες υλοποιήσεις, δημιούργησαν ένα runtime system, το οποίο λειτουργεί εν αγνοία του χρήστη και έτσι δεν χρειάζεται ο χρήστης να ανησυχεί για προγραμματιστικές λεπτομέρειες γραφικών.

Το Mars χωρίζεται σε δύο βασικά στάδια, το map στάδιο και το reduce όπως σε όλες τις υλοποιήσεις του μοντέλου. Η σειρά εκτέλεσης των διαφόρων εργασιών ακολουθεί το πρότυπο εκτέλεσης των MapReduce εργασιών, όπως έχει εξηγηθεί στο κεφάλαιο 2.3 πιο πάνω.

Η υλοποίηση αυτή είναι πολύ απλή στη χρήση. Κάποιος που θέλει να γράψει μια εφαρμογή το μόνο που χρειάζεται είναι να γράψει τις απαραίτητες συναρτήσεις

σύμφωνα με το API (Application Programmer Interface) του Mars, οι οποίες είναι user-defined. Δεν χρειάζεται να ξέρει πως γίνετε ο προγραμματισμός σε κάρτες γραφικών, αφού για αυτά ανησυχεί το runtime system.

Για τα πειράματα είχαν υλοποιηθεί κάποιες από τις εφαρμογές που είχαν υλοποιηθεί για την υλοποίηση Phoenix, οι οποίες στη συνέχεια ήρθαν σε σύγκριση μεταξύ τους. Σε όλες τις περιπτώσεις έδωσε καλύτερη επίδοση η υλοποίηση Mars. Για δεδομένα εισόδου με πολύ μεγάλο όγκο το Mars είχε speed up μέχρι και 1.5X περισσότερο από το Phoenix. Επίσης σε εφαρμογές με πολλούς υπολογισμούς το Mars παρουσιαζόταν μέχρι και 4X πιο γρήγορο από το Phoenix, ενώ σε εφαρμογές οι οποίες απαιτούν περισσότερο χρήση μνήμης η υλοποίηση Mars είναι ελάχιστα πιο γρήγορη από το Phoenix.

2.4.4 Phoenix: Evaluating MapReduce for Multi-core and Multiprocessor Systems

Είναι η υλοποίηση που έγινε για το προγραμματιστικό μοντέλο MapReduce πάνω σε shared memory συστήματα [2]. Περιλαμβάνει το programming API και το runtime system όπως και οι υπόλοιπες υλοποιήσεις που έχουν γίνει. Όπως και σε όλες τις υπόλοιπες υλοποιήσεις του προγραμματιστικού μοντέλου MapReduce, υπάρχουν οι δύο βασικές συναρτήσεις map και reduce τις οποίες πρέπει να καθορίσει ο χρήστης.

Επίσης ο χρήστης δεν χρειάζεται να έχει γνώσεις παράλληλου προγραμματισμού αφού το runtime system θα αναλάβει τέτοια ζητήματα. Συγκεκριμένα θα δημιουργήσει τα διάφορα threads που θα χρειαστούν κατά την εκτέλεση, θα χρονοδρομολογήσει δυναμικά τις εργασίες που θα εκτελεστούν. Επίσης θα αναλάβει να σπάσει τα δεδομένα σε διάφορα κομμάτια και τέλος θα χειριστεί πιθανά σφάλματα στους διάφορους επεξεργαστές που συμμετέχουν στην εκτέλεση.

Κατά την έρευνα που είχα διεξάγει για τη σχετική δουλειά πάνω στην υλοποίηση του Phoenix, δεν έχω βρει ανάλογη δουλειά με μετρήσεις της θερμοκρασία ή της κατανάλωσης ενέργειας που παρατηρείται στα συστήματα πολυεπεξεργαστών. Το συγκεκριμένο θέμα έχει απασχολήσει μέχρι τώρα τις υλοποιήσεις του προγραμματιστικού μοντέλου MapReduce που έγιναν πάνω σε συστάδες υπολογιστών

(clusters) και ειδικά πάνω στην υλοποίηση του Hadoop [3,6] αλλά όχι την υλοποίηση πάνω στα συστήματα πολυεπεξεργαστών.

Θεωρώ ότι ένα τέτοιο ζήτημα είναι πολύ σημαντικό στην προσπάθεια που γίνεται να προωθηθεί το νέο αυτό προγραμματιστικό μοντέλο. Παρόλο που δίνεται περισσότερη βαρύτητα στην επίδοση του νέου προγραμματιστικού μοντέλου, σε σχέση με το χρόνο εκτέλεσης που χρειάζεται κάθε εφαρμογή, είναι πολύ σημαντικό να ερευνηθεί κατά πόσο το μοντέλο οδηγεί σε σημαντική αύξηση στη θερμοκρασία του συστήματος. Είναι ένα προγραμματιστικό μοντέλο το οποίο απευθύνεται κυρίως σε μεγάλους Data-Centres και χρειάζεται να ξέρουν κατά πόσο η χρήση του μπορεί να οδηγήσει σε προβλήματα όπως η φθορά των συστημάτων τους ή η αύξηση οικονομικού κόστους λόγο περισσότερης κατανάλωσης ηλεκτρικού ρεύματος.

Κεφάλαιο 3

Phoenix

3.1 Υλοποίηση Phoenix

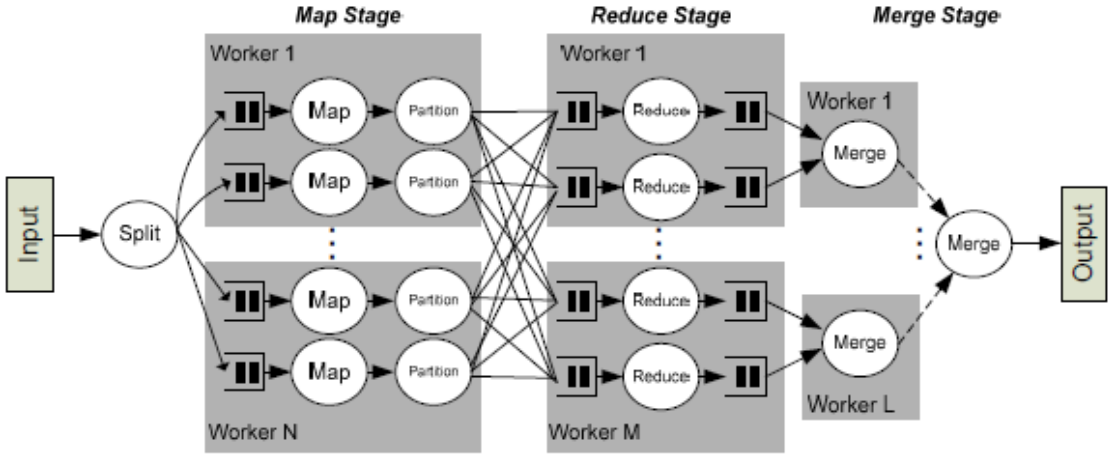
Όπως έχω ήδη αναφέρει στο κεφάλαιο 2.4.4 το Phoenix είναι η υλοποίηση που έγινε για το προγραμματιστικό μοντέλο MapReduce πάνω σε συστήματα κοινόχρηστης μνήμης[2]. Περιλαμβάνει το programming API και το runtime system όπως και οι υπόλοιπες υλοποιήσεις που έχουν γίνει για το συγκεκριμένο προγραμματιστικό μοντέλο.

Το programming API βοηθά τον χρήστη να γράψει εφαρμογές οι οποίες χρησιμοποιούν το συγκεκριμένο προγραμματιστικό μοντέλο. Χαρακτηρίζεται από δύο διαφορετικές κατηγορίες συναρτήσεων. Η πρώτη κατηγορία είναι κάποιες βασικές συναρτήσεις, οι οποίες βοηθούν την εφαρμογή να επικοινωνεί με το runtime system. Τέτοιες συναρτήσεις είναι οι phoenix_scheduler(), emit_intermediate() και emit(). Στη δεύτερη κατηγορία ανήκουν οι συναρτήσεις τις οποίες πρέπει να ορίσει ο χρήστης ανάλογα με την εφαρμογή την οποία επιθυμεί να υλοποιήσει. Οι πιο βασικές συναρτήσεις αυτής της κατηγορίας είναι και πάλι οι map και reduce συναρτήσεις. Στις δύο αυτές συναρτήσεις ο χρήστης ανάλογα με την εφαρμογή που επιθυμεί να υλοποιήσει, καλείται να καθορίσει τους υπολογισμούς και την επεξεργασία που θα τυγχάνουν τα δεδομένα εισόδου με σκοπό να παραχθεί το τελικό αποτέλεσμα.

Όσον αφορά το runtime system, είναι υπεύθυνο να δημιουργήσει τα νήματα (threads) που θα χρειαστούν, να χρονοδρομολογεί δυναμικά τις εργασίες που θα χρειαστεί να

εκτελεστούν, να σπάζει τα δεδομένα σε διάφορα κομμάτια και τέλος να χειρίζεται περιπτώσεις, στις οποίες κάποιος κόμβος μπορεί να καταρρεύσει πριν ολοκληρώσει τη δουλειά που του έχει ανατεθεί. Το runtime system ουσιαστικά ελέγχεται από τον scheduler ο οποίος μέσω του runtime system θα αναλάβει να δημιουργήσει τα threads, όπως αναφέρουμε πιο πάνω. Τα threads που δημιουργούνται χρησιμοποιούνται κυρίως για να εκτελέσουν τις map και τις reduce εργασίες, ενώ γίνεται χρήση shared memory buffers έτσι ώστε να διευκολύνεται η επικοινωνία και να μην χρειάζονται τα δεδομένα να μεταφέρονται και να αντιγράφονται από τον ένα επεξεργαστή στον άλλο. Τέλος είναι σημαντικό το γεγονός ότι οι εργασίες ανατίθενται δυναμικά στους επεξεργαστές και έτσι επιτυγχάνεται load balancing, ενώ αυξάνεται το task throughput με αποτέλεσμα οι εργασίες να τελειώνουν πιο γρήγορα.

Μια εφαρμογή για να εκτελεστεί με βάση την υλοποίηση του Phoenix, θα χρειαστεί να περάσει από τρία βασικά στάδια το map Stage, το reduce Stage και το merge Stage. Πριν την εκτέλεση των τριών βημάτων τα δεδομένα πρέπει να περάσουν από το split stage για το διαχωρισμό τους. Στο σχήμα 2.2 φαίνεται η ροή της εκτέλεσης μιας εφαρμογής όταν υλοποιηθεί με βάση Phoenix, και στη συνέχεια υπάρχουν οι εξηγήσεις σε λεπτομέρεια.



Σχήμα 2.2 Ροή Εκτέλεσης Εφαρμογής με βάση την υλοποίηση Phoenix [2]

Split Stage: Αρχικά θα κληθεί ο splitter, ο οποίος θα αναλάβει τα δεδομένα εισόδου, τα οποία θα σπάσει σε κομμάτια ίσου μεγέθους. Τα κομμάτια αυτά θα δοθούν στο επόμενο στάδιο στις map εργασίες για επεξεργασία.

Map Stage: Ο scheduler σε αυτό το στάδιο δυναμικά αναθέτει τις map εργασίες στα worker threads των επεξεργαστών που είναι διαθέσιμοι. Ο splitter με τη σειρά του θα αναθέσει στον επεξεργαστή το τμήμα των δεδομένων για την εκτέλεση της map εργασίας, κατά την οποία θα παραχθεί και θα δεσμευτεί ένα ζεύγος (κλειδί/τιμή). Εδώ υπάρχει ένα υπόβημα, το *partition*, το οποίο αναλαμβάνει κάθε ζεύγος που παράγεται. Με τα ζεύγη που αναλαμβάνει δημιουργεί ομάδες από ενδιάμεσα ζεύγη, τα οποία έχουν το ίδιο κλειδί. Έτσι γίνεται κάποιου είδους sorting, ενώ βοηθά το reduce stage να γλιτώσει extra overhead, χρόνο που θα χρειαζόταν για να ταξινομήσει όλα τα ζεύγη.

Reduce Stage: Όταν τελειώσουν όλες οι map εργασίες και παραχθούν ζεύγη (κλειδί/τιμή) για όλα τα δεδομένα εισόδου, αρχίζει το στάδιο reduce. Και εδώ οι εργασίες ανατίθενται δυναμικά όπως στο προηγούμενο στάδιο. Οι ομάδες που έχουν δημιουργηθεί κατά το partitioning, θα βοηθήσουν ώστε κάθε εργασία να αναλαμβάνει ζεύγη που έχουν ίδιο κλειδί. Θα συγχωνευθούν οι τιμές τους ανάλογα με την εφαρμογή και θα παραχθεί το τελικό αποτέλεσμα για κάθε εργασία, το οποίο θα έχει το πλεονέκτημα να είναι ταξινομημένο ως προς τα κλειδιά.

Merge stage: Το τελευταίο στάδιο είναι το στάδιο κατά το οποίο όλα τα αποτελέσματα από κάθε reduce εργασία θα συγχωνευθούν και θα δημιουργήσουν ένα τελικό αποτέλεσμα. Και πάλι η συγχώνευση είναι ανάλογα με την εφαρμογή. Υπάρχουν περιπτώσεις που θα εφαρμοστεί κάποιος υπολογισμός πάνω στα αποτελέσματα κάθε εργασίας για να παραχθεί το τελικό αποτέλεσμα και περιπτώσεις στις οποίες τα αποτελέσματα αυτά απλά θα τοποθετηθούν στο κοινό buffer για να δοθούν σαν έξοδος, χωρίς να γίνει καμιά αλλαγή.

Οι εφαρμογές οι οποίες υλοποιήθηκαν στο Phoenix, υλοποιήθηκαν και με Pthreads ώστε να υπάρχει μέτρο σύγκρισης. Η σύγκριση τους έδειξε να έχουν παρόμοιο speed

up και σε μερικές μόνο περιπτώσεις εμφανίζεται το Phoenix να έχει λίγο καλύτερο speed up. Μερικά αποτελέσματα του Phoenix, οφείλονται στο γεγονός ότι λειτουργεί με pointers, χωρίς να χρειάζεται να αντιγράφει όλα τα δεδομένα που χρειάζεται κάθε φορά, κάτι το οποίο οδηγεί σε μείωση του overhead. Επίσης, σημαντικό ρόλο στην επίδοση των εφαρμογών του Phoenix, παίζει το πως οι ίδιες οι εφαρμογές θα χειριστούν το δυναμικό scheduling που τους παρέχεται, σε αντίθεση με τις εφαρμογές σε Pthreads οι οποίες χειρίζονται το scheduling στατικά αυξάνοντας έτσι το overhead.

3.2 Phoenix Setup

Η συγκεκριμένη υλοποίηση αρχικά δημιουργήθηκε για να τρέχει σε περιβάλλον SOLARIS. Στη συνέχεια όμως έγιναν αλλαγές στην αρχική υλοποίηση, δημιουργώντας την $2^{η}$ έκδοση του Phoenix. Στην $2^{η}$ έκδοση, είχαν προσθέσει τη δυνατότητα να τρέχει η υλοποίηση και σε περιβάλλον LINUX.

Αρχικά ο χρήστης πρέπει να κατεβάσει το πακέτο του Phoenix [7] μαζί με κάποια input datasets. Δεν χρειάζεται να γίνει οποιαδήποτε εγκατάσταση, αφού η υλοποίηση του runtime system είναι ήδη μεταγλωττισμένη και μπορεί να χρησιμοποιηθεί για την εκτέλεση κάποιας εφαρμογής, η οποία θα επικοινωνεί με το runtime system μέσω του API. Στο πακέτο υπάρχουν ήδη υλοποιημένες 7 εφαρμογές, τις οποίες μπορεί ο χρήστης να τρέξει και να πάρει αποτελέσματα με τη βοήθεια των αντίστοιχων input datasets. Το μόνο που έχει να κάνει είναι να τρέξει την εφαρμογή μέσω των αντίστοιχων εντολών.

Στην περίπτωση που ο χρήστης επιθυμεί να φτιάξει τις δικές του εφαρμογές, τότε θα χρησιμοποιήσει τις συναρτήσεις που του παρέχει το API. Κάποιες συναρτήσεις θα χρειαστεί να οριστούν από το ίδιο τον χρήστης. Κάποιες άλλες είναι ήδη υλοποιημένες και με τη χρήση τους μέσα στην εφαρμογή δηλώνουν τα σημεία στα οποία το API επικοινωνεί με το runtime system και ποια ακριβώς διαδικασία πρέπει να εκτελεστεί.

3.3 Υλοποιημένες Εφαρμογές

Οι φοιτητές του Stanford [7], για να μπορέσουν να αξιολογήσουν την υλοποίηση του Phoenix, χρειάστηκε να υλοποιήσουν 7 εφαρμογές, οι οποίες να ακολουθούν το προγραμματιστικό μοντέλο MapReduce, και τις αντίστοιχες εφαρμογές σε υλοποίηση Phtreads. Οι 7 αυτές εφαρμογές ανήκαν σε τέσσερις κατηγορίες υπολογισμών, α) enterprise computing, β) scientific computing, γ) artificial intelligent και δ) image processing. Στη συνέχεια, επεξηγούνται οι εφαρμογές που υπάρχουν στο πακέτο του Phoenix.

Word Count: Εφαρμογή κατά την οποία υπολογίζεται η συχνότητα εμφάνισης κάθε λέξης σε ένα σύνολο αρχείων.

Matrix Multiply: Υλοποιεί αλγόριθμο για πολλαπλασιασμό πινάκων ακεραίων αριθμών.

String Match: Σε αυτή την εφαρμογή δίνεται ένα αρχείο με κάποιες λέξεις κλειδιά και ένα άλλο αρχείο κειμένου, στο οποίο η εφαρμογή θα ψάξει να βρει τις λέξεις που ταιριάζουν με τις λέξεις κλειδιά.

Kmeans: Υλοποιεί τον Kmeans αλγόριθμο, ο οποίος ομαδοποιεί ένα σύνολο δεδομένων εισόδου σε clusters.

PCA: Χρησιμοποιεί τον αλγόριθμο Principal Component Algorithm. Αυτό που προσπαθεί να κάνει είναι να βρει ένα διάνυσμα το οποίο να κυμαίνεται ανάμεσα στις τιμές του πίνακα κάποιου συνόλου δεδομένων.

Histogram: Αναλύει μια εικόνα τύπου bitmap, και υπολογίζει τη συχνότητα ύπαρξης, RGB components στα pixels με τιμή ανάμεσα σε 0-255.

Linear Regression: Υπολογίζει την γραμμή που ταιριάζει καλύτερα σε ένα σύνολο συντεταγμένων οι οποίες δίνονται σαν δεδομένα εισόδου.

Αφού έτρεξαν τις εφαρμογές που ήταν γραμμένες σε κώδικα Phoenix, τις είχαν συγκρίνει με τις αντίστοιχες εφαρμογές σε κώδικα Pthreads. Από αυτή τη σύγκριση κατέληξαν στο συμπέρασμα ότι οι επίδοση των εφαρμογών σε Phoenix ήταν παρόμοια με την επίδοση των αντίστοιχων εφαρμογών σε Pthreads.

Κεφάλαιο 4

Phoenix και Θερμοκρασία

4.1 Πολυπύρηνοι Επεξεργαστές: Αύξηση επίδοσης και Θερμοκρασία

Με τη νέα αρχιτεκτονική που προωθείται στην αγορά ουσιαστικά σηματοδοτείται μια νέα εποχή σε ερευνητικό επίπεδο. Οι πολυπύρηνοι επεξεργαστές από μόνοι τους δεν μπορούν να δώσουν καλύτερη επίδοση. Τα προγράμματα που γράφονταν μέχρι αυτό το σημείο ήταν σχεδιασμένα για να τρέχουν σειριακά, χρησιμοποιώντας μόνο ένα επεξεργαστή. Για να μπορέσουμε να εκμεταλλευτούμε τα νέα συστήματα πολυεπεξεργαστών έπρεπε να αρχίσουν να γράφονται προγράμματα που να χρησιμοποιούν αποδοτικά τον παραλληλισμό. Επίσης πρέπει τα προγράμματα αυτά να μπορούν να προσαρμόζονται από μικρό αριθμό πυρήνων σε αρκετά μεγαλύτερο αριθμό πυρήνων και να συνεχίζουν να διατηρούν την καλή τους επίδοση. Όλες οι προσπάθειες που γίνονται οδηγούν, όπως αναφέρω στο Κεφάλαιο 2.2, στη δημιουργία νέων μοντέλων προγραμματισμού.

Στα διάφορα μοντέλα, οι επεξεργαστές που ακολουθούν τη συγκεκριμένη αρχιτεκτονική καλούνται να εκτελέσουν λειτουργίες με μεγάλο υπολογιστικό φόρτο. Ο μεγάλος υπολογιστικός όγκος οδηγεί συχνά στην αύξηση της θερμοκρασίας στο σύστημα. Αυτό οφείλεται και πάλι στο γεγονός ότι τα κυκλώματα στο κύκλωμα (chip) εξελίσσονται και γίνονται πιο περίπλοκα. Αρχικά τοποθετήθηκαν περισσότερα απλά κυκλώματα σε ένα κύκλωμα (chip), αλλά η εξέλιξη αναγκάζει τις νέες τεχνολογίες και πάλι να γίνονται πιο περίπλοκες. Το αποτέλεσμα της αύξησης της πολυπλοκότητας στα κυκλώματα, οδηγεί και πάλι στην κατανάλωση περισσότερης ισχύς η οποία

μετατρέπεται σε θερμότητα. Λαμβάνοντας υπόψη ότι η επιφάνεια των πυρήνων συνεχίζει να είναι η ίδια, τότε η ισχύς που καταναλώνεται η οποία μετατρέπεται σε θερμότητα έχει λιγότερο χώρο να εξαπλωθεί και έτσι παρατηρείται αύξηση στην θερμοκρασία του συστήματος.

4.2 Θερμοκρασία σε Κέντρα Δεδομένων

Η θερμοκρασία όταν παρουσιάζεται σε ψηλές τιμές στα διάφορα υπολογιστικά συστήματα, υπονοεί την ύπαρξη κάποιων προβλημάτων. Εάν η θερμοκρασία στους πυρήνες δεν κρατηθεί στα επιθυμητά επίπεδα, ίσως προκαλέσει κάποια βλάβη στο υλικό του συστήματος, κάτι το οποίο έμμεσα μειώνει την επίδοση του συστήματος. Επίσης, αφού η αύξηση στη θερμοκρασία παρατηρείται από την αυξημένη ισχύ που χρειάζονται τα διάφορα κυκλώματα, συνεπάγεται και αύξηση της ηλεκτρικής ενέργειας.

Το θέμα της Θερμοκρασίας, είναι ένα πολύ μεγάλο ζήτημα, το οποίο απασχολεί τα μοντέρνα κέντρα δεδομένων (data centres), αφού υψηλές τιμές στη θερμοκρασία όπως έχω ήδη αναφέρει μεταφράζεται σε υψηλό κόστος κατανάλωσης ηλεκτρικού ρεύματος. Καθώς παρατηρείται τελευταία, ότι πολλές υπηρεσίες στο διαδίκτυο στηρίζουν την λειτουργία τους πάνω στο προγραμματιστικό μοντέλο MapReduce, υπάρχει ανάγκη να γνωρίζουν κατά πόσο το μοντέλο αυτό μπορεί να τους προσφέρει καλή επίδοση και ταυτόχρονα χαμηλό κόστος κατανάλωσης ηλεκτρικού ρεύματος.

Στη συγκεκριμένη εργασία, έχω αναλάβει να πάρω τις κατάλληλες μετρήσεις στη θερμοκρασία των πυρήνων όταν χρησιμοποιείται το προγραμματιστικό μοντέλο MapReduce μέσω της υλοποίησης του Phoenix. Όπως έχω αναφέρει στο κεφάλαιο 3.3 στο πακέτο της υλοποίησης του Phoenix υπάρχουν κάποιες εφαρμογές υλοποιημένες. Σκοπός ήταν να τρέξω τις εφαρμογές αυτές και κατά τη διάρκεια της εκτέλεσης τους, με τη βοήθεια κάποιου εργαλείου μέτρησης της θερμοκρασίας να παίρνω τις κατάλληλες μετρήσεις. Σε αυτό το βήμα, με έχει βοηθήσει το εργαλείο Lm_sensors το οποίο εξηγώ στο αμέσως επόμενο κεφάλαιο. Συγκεκριμένα, έχω ενσωματώσει το εργαλείο αυτό, μέσα στον κώδικα των εφαρμογών του Phoenix. Με αποτέλεσμα κατά

την εκτέλεση της εφαρμογής να καταγράφονται οι τιμές στη θερμοκρασία όλων των πυρήνων που χρησιμοποιούνται στη συγκεκριμένη εκτέλεση.

Επίσης χρειάστηκε να κάνω και κάποιες αλλαγές στον κώδικα των εφαρμογών που ήταν ήδη υλοποιημένες, έτσι ώστε να μπορεί ο χρήστης που θα τρέξει κάποια εφαρμογή, να ορίζει τον αριθμό των πυρήνων που επιθυμεί να χρησιμοποιήσει. Η μηχανή, η οποία χρησιμοποίησα για τα πειράματα μου αποτελείται από 8 πυρήνες. Με τη συγκεκριμένη αλλαγή, κατάφερα να τρέχω την ίδια εφαρμογή χρησιμοποιώντας 1 πυρήνα, 2, 4 και 8.

4.3 Lm_Sensors: Linux hardware monitoring

Το Lm_sensors [9] είναι το εργαλείο που χρησιμοποίησα στην εργασία μου για να παίρνω τις απαραίτητες μετρήσεις. Ουσιαστικά είναι ένα εργαλείο το οποίο χειρίζεται διάφορα ζητήματα που αφορούν το hardware health status της μηχανής σε περιβάλλον LINUX, με τη βοήθεια των kernel drivers και των user-space libraries. Προσφέρεται δωρεάν σε όλους τους LINUX users, αφού είναι ένα open source software-tool. Έχει τη δυνατότητα να παρέχει πληροφορίες, από δεδομένα που αποκτά από το υλικό της μηχανής. Ανιχνεύει το περιβάλλον στο υλικό της μηχανής, ενώ εμφανίζει στο χρήστη δεδομένα που του δίνουν οι διάφοροι sensors. Ανάμεσα στις πολλές πληροφορίες που δίνει το Lm_sensors, είναι και οι πληροφορίες σχετικά με τα voltages που καταναλώνονται, το fan speed και τη θερμοκρασία στους πυρήνες του συστήματος.

Ιστορικά οι πρώτοι αισθητήρες που τοποθετήθηκαν στο υλικό των διαφόρων συστημάτων ανήκαν στην κατηγορία των αναλογικών αισθητήρων. Τους αισθητήρες αυτούς ήρθαν να αντικαταστήσουν στη συνέχεια οι ψηφιακοί αισθητήρες (DTS – Digital Thermal Sensors), οι οποίοι είναι πολύ πιο μικροί σε μέγεθος, κάτι που τους βοηθά να τοποθετούνται πιο κοντά στο hot spot. Έτσι μπορούν να καταμετρούν τη θερμοκρασία πιο γρήγορα και με περισσότερη ακρίβεια.

Στα σημερινά συστήματα υπολογιστών, είναι τοποθετημένο ένα manager chip, το οποίο διαβάζει τις πληροφορίες από τις τιμές της θερμοκρασίας που δίνουν οι αισθητήρες. Με βάση αυτές τις πληροφορίες θα προσπαθήσει να επηρεάσει το

περιβάλλον του συστήματος ελέγχοντας τα fans του. Η ταχύτητα των fans (fan speed) μπορεί να αυξηθεί ή να μειωθεί είτε από το χρήστη, είτε αυτόματα από το σύστημα. Παρόλη την τεχνολογία που παρέχεται όμως από τα διάφορα συστήματα, ο χρήστης δεν μπορεί να είναι σίγουρος ότι οι ρυθμίσεις στο fan speed παρέχουν αρκετή ροή αέρα ώστε να κρατά τα εξαρτήματα του συστήματος ικανοποιητικά κρύα.

Για τη συγκεκριμένη εργασία είχα στη διάθεση μου ένα πολυπύρηνο υπολογιστικό σύστημα, το οποίο αποτελείται από 2 Quad core Intel Xeon E5320 επεξεργαστές. Όπως αναφέρει η Intel οι επεξεργαστές του συγκεκριμένου τύπου [11], έχουν ανάγκη από θέματα διαχείρισης θερμότητας. Τέτοιοι επεξεργαστές αποτελούνται από μεγάλα και πολύπλοκα κυκλώματα, τα οποία πρέπει να είναι σε θέση να διασφαλίζουν την αξιοπιστία του συστήματος τους.

Έτσι η εταιρεία κατά τον σχεδιασμό και την κατασκευή των συγκεκριμένων επεξεργαστών έχει εφαρμόσει κάποιο thermal monitor για προστασία του επεξεργαστή. Έχει λάβει επίσης υπόψη της ότι ο καλύτερος τρόπος διαχείρησης θερμότητας εξαρτάται από την τοποθέτηση ψηκτρών κοντά στον επεξεργαστή, όπως επίσης και από τον τρόπο που τοποθετούνται τα διάφορα εξαρτήματα έτσι που να υπάρχει καλύτερη ροή αέρα μέσα από αυτά και ουσιαστικά να γίνεται καλύτερος εξαερισμός.

Όσον αφορά τους αισθητήρες που έχουν τοποθετηθεί σε αυτού του είδους επεξεργαστές, η εταιρεία δεν δίνει σχετικές πληροφορίες για τα σημεία τοποθέτησης τους. Στη ιστοσελίδα [11] της όμως, αναφέρεται στους αισθητήρες θερμότητας που υπάρχουν στον επεξεργαστή τονίζοντας ότι διαβάζουν τις τιμές της θερμοκρασίας των πυρήνων και στη συνέχεια τις εξάγει μέσω του System Management Bus (SMBus). Οι πληροφορίες που αφορούν την τιμή της θερμοκρασίας κάθε πυρήνα, ονομάζονται “Thermal Bytes” έχουν μέγεθος 8-bits, και μπορούν να διαβαστούν ανά πάσα στιγμή από τον αντίστοιχο Thermal Sensor. Στις τιμές που διαβάζονται πιθανόν να υπάρχει λάθος στην μέτρηση μέχρι 1°C, όχι περισσότερο.

Επίσης, στον επεξεργαστή έχει αποθηκευτεί κατά την κατασκευή του κάποια πληροφορία μεγέθους 8-bits και έχει ονομαστεί Thermal Reference Byte. Το Thermal Reference Byte είναι διαθέσιμο μέσω του Processor Information ROM στο SMBus και

αντιπροσωπεύει τη μέγιστη τιμή που μπορεί να πάρει ο επεξεργαστής όταν υπερφορτώνεται. Αν καμιά φορά ο επεξεργαστής ξεπεράσει τα συγκεκριμένα επίπεδα θερμοκρασίας, όπως ορίζονται από το Thermal Reference Byte, συνεχίζει να τρέχει παρόλο που είναι πιο θερμός και υπάρχει πιθανότητα να αρχίσουν να παρουσιάζονται φθορές στο σύστημα.

Σε αυτό το σημείο είναι που έρχεται να βοηθήσει το software. Έχουν δημιουργηθεί διάφορα εργαλεία, τα οποία διαβάζουν τις τιμές της θερμοκρασίας από τους thermal Sensors του συστήματος, ανάμεσα τους και το Lm_sensors. Στη συνέχεια συγκρίνουν την τιμή που έχουν διαβάσει από κάθε Thermal Sensor και την αντίστοιχη τιμή που δίνει το Thermal Reference Byte. Αν καταλάβει ότι ο επεξεργαστής έχει υπερφορτωθεί και η θερμοκρασία του έχει ανέβει σε επικίνδυνα επίπεδα θα τον βοηθήσει ώστε να μειωθεί η θερμοκρασία του. Τα περισσότερα εργαλεία σε τέτοια περίπτωση ανεβάζουν το fun speed για να υπάρχει περισσότερη ροή αέρα στο σύστημα ή θέτουν τους πιο ζεστούς πυρήνες εκτός λειτουργίας μέχρι να ρίξουν την θερμοκρασία σε πιο χαμηλά επίπεδα και να αρχίσουν ξανά δουλειά.

Για την εργασία μου, χρειάστηκε να χρησιμοποιήσω το συγκεκριμένο εργαλείο έτσι ώστε να μπορώ να διαβάζω τις τιμές στη θερμοκρασία των πυρήνων του συστήματος. Το Lm_sensors μπορεί να διαβάσει την τιμή στη θερμοκρασία κάθε πυρήνα από τον αντίστοιχο thermal sensor που βρίσκεται ενσωματωμένος πάνω του. Την τιμή που διαβάζει το εργαλείο αυτό την παρουσιάζει στην οθόνη του χρήστη μέσω του terminal. Χαρακτηριστικό του εργαλείου αυτού είναι ότι μπορεί να δίνει την τιμή της θερμοκρασίας, ξεχωριστά για κάθε πυρήνα. Επίσης, έχει τη δυνατότητα να ανιχνεύει την κρίσιμη τιμή της θερμοκρασίας των πυρήνων (Thermal Reference Byte) και στην περίπτωση που η θερμοκρασία κάποιου πυρήνα φτάσει στο Thermal Reference Byte σημείο της, τότε το εργαλείο από μόνο του αυτόματα θα αναλάβει να θέσει εκτός λειτουργίας τον συγκεκριμένο πυρήνα, μέχρι να χαμηλώσει η θερμοκρασία του για να μπορεί να τεθεί ξανά σε λειτουργία. Δεν χρειάζεται ο χρήστης του συστήματος να αναλάβει να κάνει κάτι αντίστοιχο.

Κεφάλαιο 5

Εφαρμογές TCP-H

5.1 Περιγραφή TCP-H

Το TPC – Transaction Processing Performance Council είναι ένας μη κερδοσκοπικός οργανισμός [10], ο οποίος έχει σαν σκοπό του να ορίσει την επεξεργασία συναλλαγών και κάποια benchmarks βάσεων δεδομένων τα οποία αφορούν κυρίως τη βιομηχανία. Στη συνέχεια προσπαθεί με αντικειμενικότητα να διαδίδει την επίδοση των TPC δεδομένων στη βιομηχανία και δίνει λύσεις σε σημαντικά ερωτήματα που αφορούν κυρίως την ανταλλαγή προϊόντων, τις υπηρεσίες και το κόστος.

Ανάμεσα στα benchmarks που προωθεί το TPC βρίσκεται το TPC-H benchmark. Η συγκεκριμένη κατηγορία benchmarks αποτελείται από benchmarks υποστήριξης απόφασης. Ουσιαστικά απεικονίζει συστήματα υποστήριξης απόφασης (decision support systems), τα οποία έχουν να επεξεργαστούν μεγάλο όγκο δεδομένων, να εκτελέσουν queries με μεγάλο βαθμό πολυπλοκότητας και να δώσουν απαντήσεις σε κρίσιμα ερωτήματα στις επιχειρήσεις.

5.2 Περιγραφή Queries

Για να γνωρίσω καλύτερα το προγραμματιστικό μοντέλο MapReduce, επέλεξα να υλοποιήσω το query 6 κάνοντας χρήση των υπηρεσιών που παρέχει το API της υλοποίησης του Phoenix. Το query 6, ανήκει στο TPC-H Benchmark Suite και έχει σχέση με queries τα οποία απαντούν σε ερωτήματα διαφόρων εταιριών και

οργανισμών. Η συγκεκριμένη επιλογή έγινε λόγω του μεγάλου όγκου δεδομένων τα οποία δίνονται σαν είσοδος για την εκτέλεση του συγκεκριμένου query.

Αυτό το query υπολογίζει την αύξηση των εσόδων τα οποία προκύπτουν από τον αποκλεισμό ορισμένου μέρους από το ποσοστό εκπτώσεων στις εταιρείες μέσα σε ορισμένο χρονικό διάστημα [10].

Το Forecasting Revenue Change (Q6) query εξετάζει όλα τα lineitems, τα οποία αποστέλλονται σε ένα συγκεκριμένο χρονικό διάστημα, στα οποία εφαρμόζεται έκπτωση από DISCOUNT-0.01 μέχρι DISCOUNT+0.01. Το query θα απαριθμήσει το ποσό με το οποίο το συνολικό εισόδημα θα είχε αυξηθεί εάν είχαν αποκλειστεί εκπτώσεις στα lineitems με l_quantity λιγότερη από quantity. Η πιθανή αύξηση των εσόδων ισούται με το ποσό [l_extendedprice * l_discount] για όλα τα lineitems με τις εκπτώσεις και τις ποσότητες στην κατάλληλη σειρά.

Στο παράρτημα Α.2, φαίνεται ο ψευδοκώδικας του συγκεκριμένου query σε SQL.

5.3 Υλοποίηση σε MapReduce

Για την υλοποίηση του query 6 στο Phoenix, αρχικά έπρεπε να διαβάζω το αρχείο με τα δεδομένα εισόδου. Για κάθε record του αρχείου που διάβαζα είχα ένα πίνακα struct, όπου τοποθετούσα για κάθε record τα στοιχεία που χρειάζονται για την εκτέλεση του query. Στη συνέχεια με την συνάρτηση map_reduce_init() δήλωνα ότι αρχίζει η εκτέλεση της εφαρμογής με βάση το μοντέλο MapReduce, ουσιαστικά γινόταν κλήση του scheduler.

Στη συνέχεια γινόταν ορισμός των MapReduce arguments, μέσα από τα οποία έστελνα στην συνάρτηση map, τα δεδομένα εισόδου, ενώ μπορούσα να ορίζω τον αριθμό των πυρήνων που θα χρησιμοποιούσε σε κάθε περίπτωση η εφαρμογή.

Μετά τον ορισμό των MapReduce arguments, γινόταν κλήση της συνάρτησης q6_map(), της συνάρτησης q6_combiner() και τέλος της συνάρτησης q6_reduce().

Όταν τελείωνε η MapReduce εκτέλεση δήλωνα το τέλο ς της με τη συνάρτηση map_reduce_finalize().

Πριν τη συνάρτηση q6_map, τα δεδομένα εισόδου μοιράζονταν σε διάφορα κομμάτια και δίνονταν για εκτέλεση στους πυρήνες τους συστήματος. Εκεί ουσιαστικά κάθε map εργασία έπαιρνε ένα ένα τα δεδομένα εισόδου (κάθε δεδομένο αντιπροσωπεύει ένα record) όριζε σαν τιμή του ζεύγους το αποτέλεσμα του υπολογισμού ext_price*year και σαν κλειδί μια τυχαία τιμή από 1 – 10. Στη συνέχεια με την emit_intermediate(), δεσμευόταν κάθε ζεύγος (κλειδί/τιμή) για να δοθεί στη συνέχεια για ταξινόμηση. Όταν τελειώνει κάθε map εργασία καλείται από το runtime system η συνάρτηση partition, η οποία θα αναλάβει να ταξινομήσει τα ζεύγη που έχουν παραχθεί από τη συγκεκριμένη εργασία.

Πριν τελειώσει αυτό το στάδια, κάθε πυρήνας ο οποίος έχει αναλάβει μια map εργασία και αφού έχουν ταξινομηθεί τα ζεύγη που έχει παράγει, για ολόκληρο το block δεδομένων που έχει αναλάβει θα καλέσει την συνάρτηση q6_combiner() ώστε στις τιμές των ζευγών που έχουν παραχθεί και έχουν κοινό κλειδί να γίνει πρόσθεση. Ουσιαστικά εδώ προσθέτονται οι ext_price*year υπολογισμοί που έγιναν κατά την map, μεταξύ τους. Τα αποτελέσματα που παράγονται, τοποθετούνται σε ενδιάμεσο buffer για να θα δοθούν στη συνέχεια από τον scheduler στους πυρήνες όταν θα τους αναθέσει τις reduce εργασίες.

Τα αποτελέσματα από την συνάρτηση q6_combiner() θα ταξινομηθούν από το runtime system, και τα ζεύγη που έχουν κοινό κλειδί θα τοποθετηθούν στην ίδια ομάδα. Κατά την ανάθεση των reduce εργασιών και την κλήση της q6_reduce(), κάθε πυρήνας αναλαμβάνει μια ομάδα με ζεύγη τα οποία έχουν κοινό κλειδί και εκτελεί τον ίδιο υπολογισμό όπως και στην συνάρτηση q6_combiner. Σε αυτή την εκτέλεση όμως έχουμε όλα τα ζεύγη με κοινό κλειδί σε μια ομάδα και έτσι παίρνουμε τελικό αποτέλεσμα για κάθε ξεχωριστό κλειδί. Τα αποτελέσματα αυτά τοποθετούνται σε ένα buffer.

Αφού ολοκληρωθεί η MapReduce εκτέλεση, χρειάζεται να αθροίσουμε τις τελικές τιμές για κάθε κλειδί έτσι ώστε να παραχθεί ένα τελικό ενιαίο αποτέλεσμα, όπως ορίζει το query 6 και τερματίζεται η εκτέλεση της εφαρμογής.

Ο κώδικας μου παρουσιάζεται στο παράρτημα Α.3.

Κεφάλαιο 6

Αποτελέσματα

6.1 Περιβάλλον εγκατάστασης

Σε αυτό το σημείο θα δώσω τα κύρια χαρακτηριστικά της μηχανής στην οποία έχω τρέξει τα πειράματα και έχω πάρει τις απαραίτητες μετρήσεις. Επίσης θα αναφέρω τα εργαλεία τα οποία έχω χρησιμοποιήσει στην εργασία μου, και τέλος τις εφαρμογές τις οποίες έτρεξα για να πάρω τις απαραίτητες μετρήσεις.

6.1.1 Χαρακτηριστικά Μηχανής

Για την εγκατάσταση των εργαλείων και την εκτέλεση των απαραίτητων εφαρμογών έχω χρησιμοποιήσει μια μηχανή IBM ServerX με 2 Intel Xeon E5320. Ουσιαστικά η μηχανή αυτή αποτελείται από οκτώ πυρήνες, συχνότητας 1,6GHz ο καθένας. Η

συγκεκριμένη μηχανή διαθέτει 16GB κύριας μνήμης και σκληρό δίσκο χωρητικότητας 500 GB.

6.1.2 Εργαλεία

Η μηχανή την οποία αναφέρω πιο πάνω τρέχει στο λειτουργικό σύστημα LINUX Ubuntu 9.04 - 64 bits. Επίσης έχει χρειαστεί να εγκαταστήσω την έκδοση 3.1.2 από το εργαλείο Lm_sensors για την καταμέτρηση των τιμών της θερμοκρασίας. Όσον αφορά την υλοποίηση του Phoenix δεν χρειάζεται να γίνει κάποια εγκατάσταση, χρειάζεται μόνο να κατέβει το πακέτο του Phoenix από την σελίδα του [7]. Ο χρήστης μπορεί να τρέξει αμέσως κάποιες εφαρμογές που εμπεριέχονται στο πακέτο και είναι ήδη μεταγλωττισμένες ή να γράψει τις δικές του εάν το επιθυμεί.

6.1.3 Εφαρμογές

Για να πάρω μετρήσεις σχετικά με τη θερμοκρασία των πυρήνων του συστήματος όταν τρέχουν MapReduce εφαρμογές, χρησιμοποίησα τις εφαρμογές του histogram, linear Regression, string match, pca και word count οι οποίες ήταν ήδη υλοποιημένες. Για τις εφαρμογές linear Regression, string match και word count πήρα μετρήσεις από διαφορετικά μεγέθη αρχείων. Στις δύο πρώτες εφαρμογές χρησιμοποίησα αρχεία μεγέθους 50MB, 100MB, 500MB και 1GB και στην τελευταία εφαρμογή χρησιμοποίησα αρχεία μεγέθους 10MB, 50MB, 100MB, 500MB και 1GB.

6.2 Εφαρμογές Phoenix

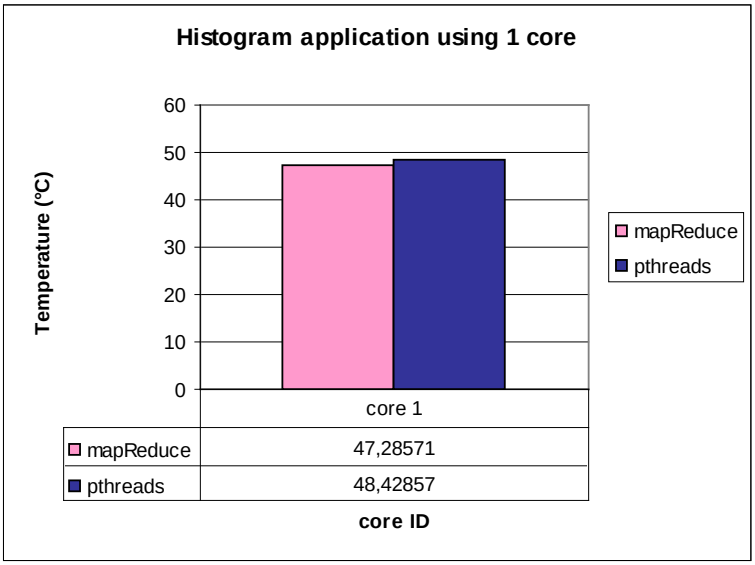
Όπως έχω ήδη αναφέρει στο Κεφάλαιο 6.1.3, έχω πάρει μετρήσεις που αφορούν τη θερμοκρασία των πυρήνων του συστήματος από της εφαρμογές histogram, linear regression, string match, pca και word count. Συγκεκριμένα έτρεχα κάθε εφαρμογή χρησιμοποιώντας διαφορετικό αριθμό πυρήνων από το σύστημα, αρχίζοντας από 1 πυρήνα, συνεχίζοντας με 2, 4 και τέλος με 8 πυρήνες καταγράφοντας κάθε φορά τις τιμές στη θερμοκρασία των πυρήνων κατά την εκτέλεση. Ακολούθως, ανάλογα με τον αριθμό των πυρήνων που χρησιμοποιήθηκαν, χρειάστηκε να υπολογίσω το μέσο όρο στις τιμές της θερμοκρασίας κάθε πυρήνα για τις 10 διαφορετικές εκτελέσεις. Οι

μεγαλύτερες και οι μικρότερες τιμές της θερμοκρασίας που καταγράφτηκαν αφαιρέθηκαν, για πιο σίγουρα αποτελέσματα.

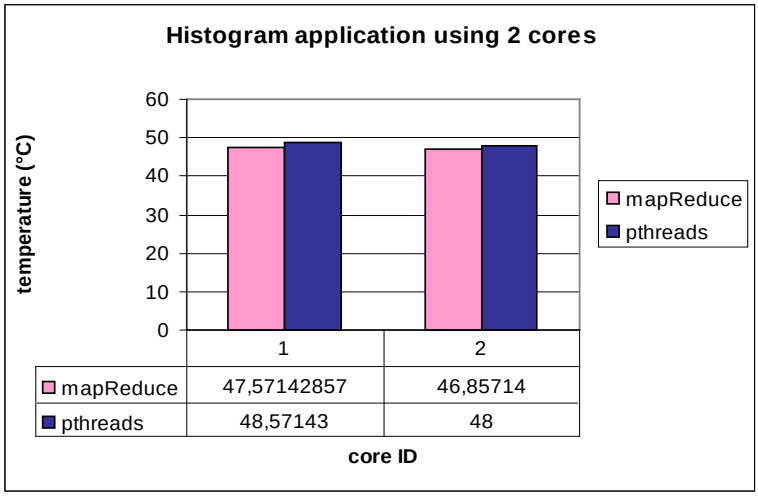
Πιο κάτω παραθέτω τα αποτελέσματα από τα πειράματα που έτρεξαν για κάθε εφαρμογή.

6.2.1 Histogram

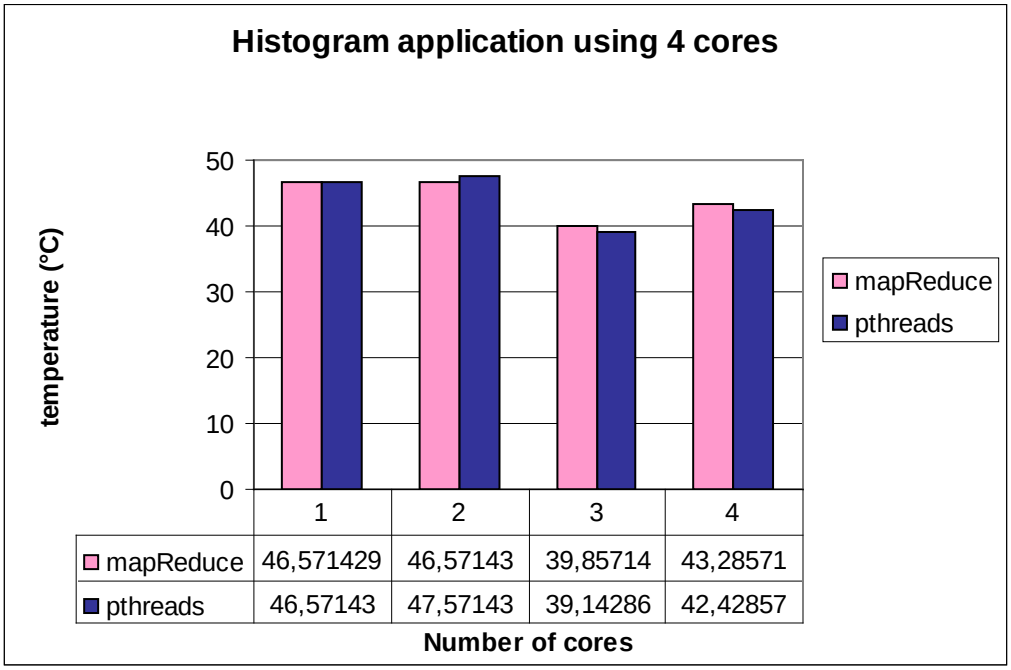
Για τα 4 σενάρια που έτρεξα (1, 2, 4 και 8 πυρήνες) χρησιμοποιήθηκε εικόνα μεγέθους 1.30 GB.



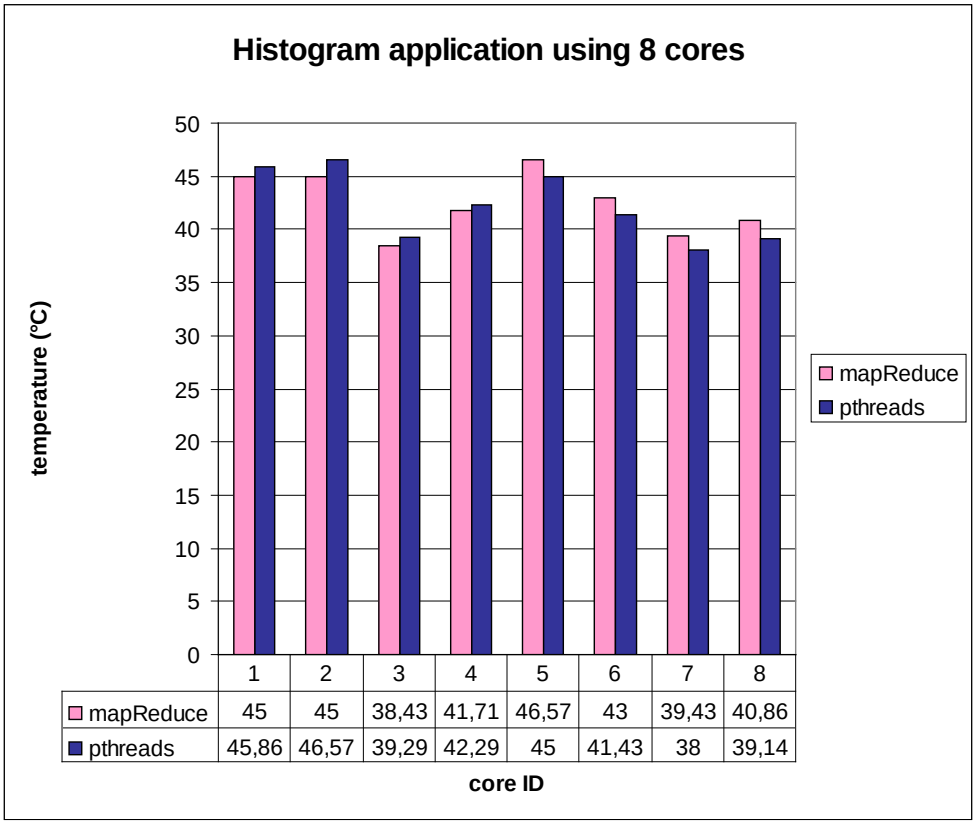
Σχήμα 6.2.1.α: Εφαρμογή Histogram που χρησιμοποιεί 1 πυρήνα του συστήματος



Σχήμα 6.2.1.β: Εφαρμογή Histogram που χρησιμοποιεί 2 πυρήνες του συστήματος



Σχήμα 6.2.1.γ: Εφαρμογή Histogram που χρησιμοποιεί 4 πυρήνες του συστήματος



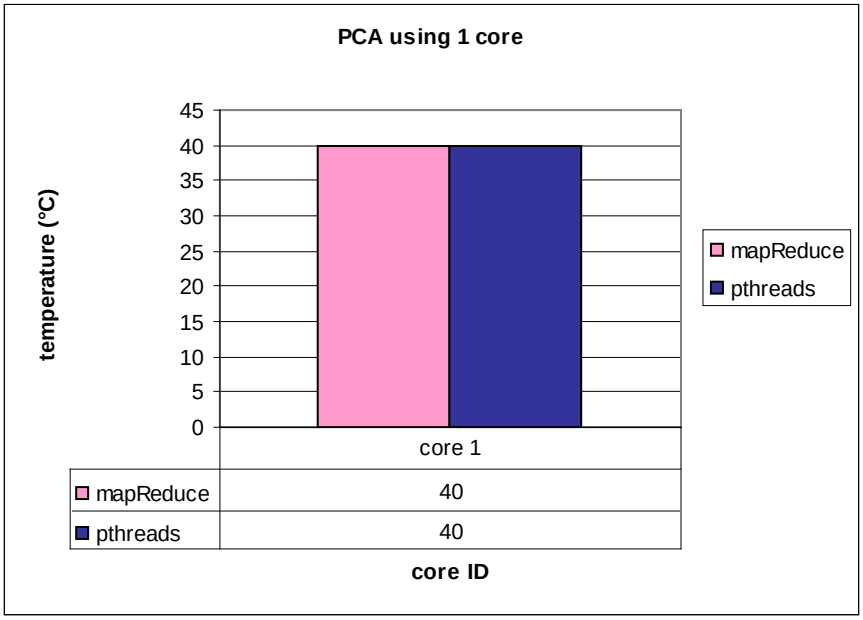
Σχήμα 6.2.1.γ: Εφαρμογή Histogram που χρησιμοποιεί 8 πυρήνες του συστήματος

Στα σχήματα 6.2.1.α, 6.2.1.β, 6.2.1.γ και 6.2.1.δ παρουσιάζεται ο μέσος όρος των τιμών στην θερμοκρασία κάθε πυρήνα όπως έχουν καταγραφεί από την εκτέλεση της

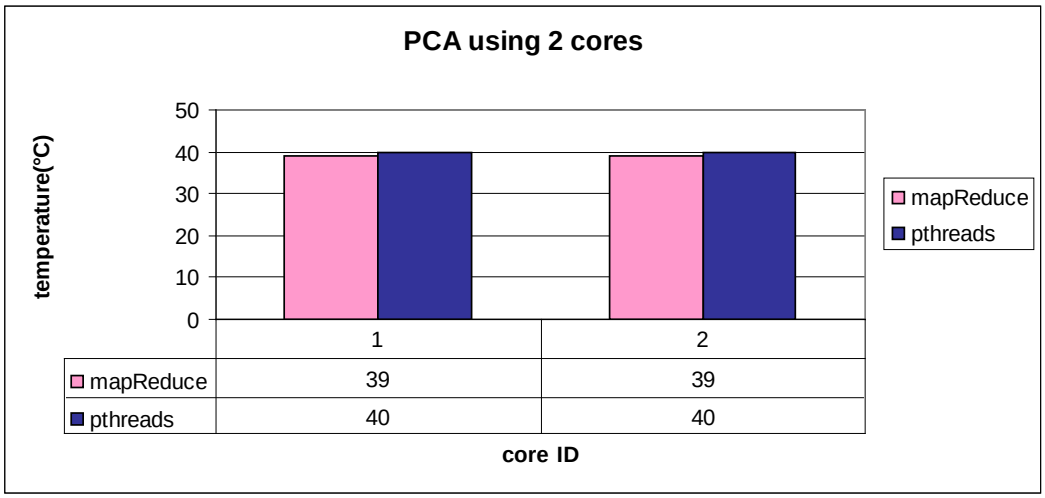
εφαρμογής histogram. Τα αποτελέσματα είναι σε σύγκριση με τα αντίστοιχα αποτελέσματα της ίδιας εφαρμογής υλοποιημένης σε pthreads και όπως παρατηρούμε δεν υπάρχουν σημαντικές διαφορές ανάμεσα στις τιμές που καταγράφηκαν στη θερμοκρασία των εφαρμογών στην υλοποίηση Phoenix και στις εφαρμογές στην υλοποίηση με pthreads.

6.2.2 PCA

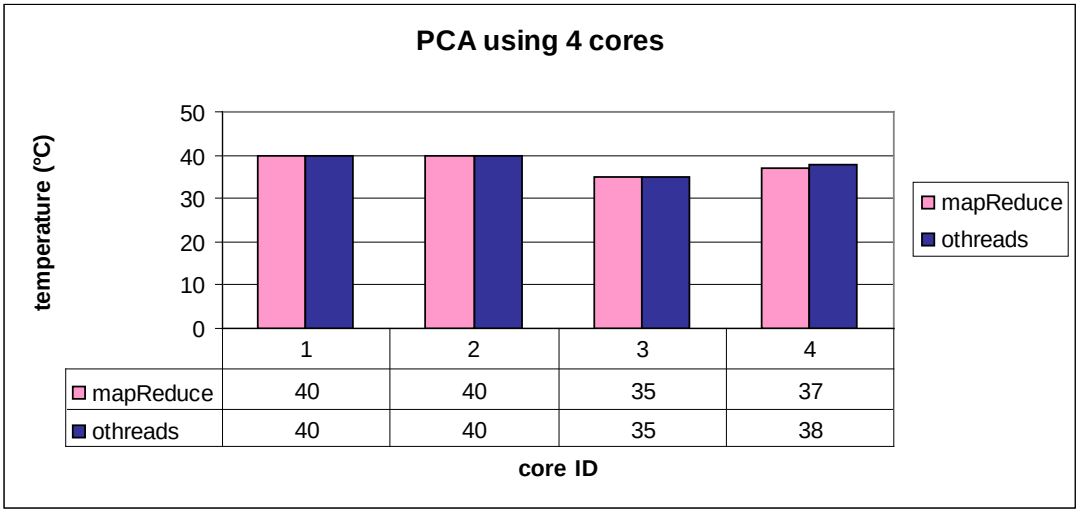
Για αυτή την εφαρμογή έτρεξα 4 σενάρια για 1, 2, 4 και 8 πυρήνες. Τα δεδομένα τα οποία χρησιμοποιούνται σαν είσοδος στην συνάρτηση map, δημιουργούνται τυχαία κατά την εκτέλεση ανάλογα με το μέγεθος του πίνακα τον οποίο ορίζει ο χρήστης όταν θα τρέξει την εφαρμογή.



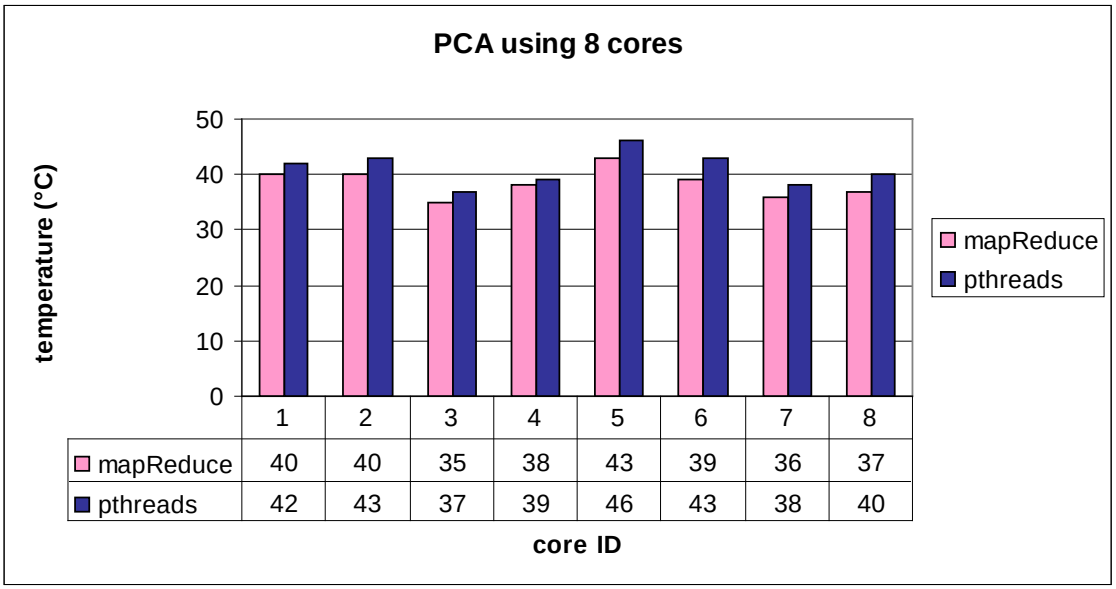
Σχήμα 6.2.2.α: Εφαρμογή PCA που χρησιμοποιεί 1 πυρήνα του συστήματος



Σχήμα 6.2.2.β: Εφαρμογή PCA που χρησιμοποιεί 2 πυρήνες του συστήματος



Σχήμα 6.2.2.γ: Εφαρμογή PCA που χρησιμοποιεί 4 πυρήνες του συστήματος



	1	2	3	4	5	6	7	8
mapReduce	40	40	35	38	43	39	36	37
pthreads	42	43	37	39	46	43	38	40

Σχήμα 6.2.2.δ: Εφαρμογή PCA που χρησιμοποιεί 8 πυρήνες του συστήματος

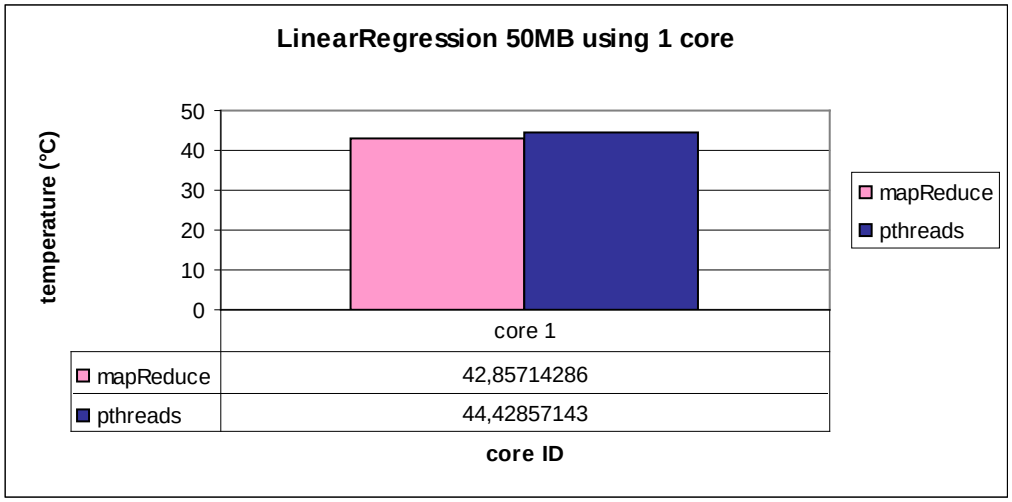
Στα σχήματα 6.2.2.α, 6.2.2.β, 6.2.2.γ και 6.2.2.δ παρουσιάζεται ο μέσος όρος των τιμών στην θερμοκρασία κάθε πυρήνα όπως έχουν καταγραφεί από την εκτέλεση της εφαρμογής PCA. Τα αποτελέσματα είναι σε σύγκριση με τα αντίστοιχα αποτελέσματα της ίδιας εφαρμογής υλοποιημένης σε pthreads. Στην περίπτωση και αυτής της εφαρμογής δεν παρατηρούμε σημαντικές διαφορές ανάμεσα στις εφαρμογές των δύο υλοποιήσεων.

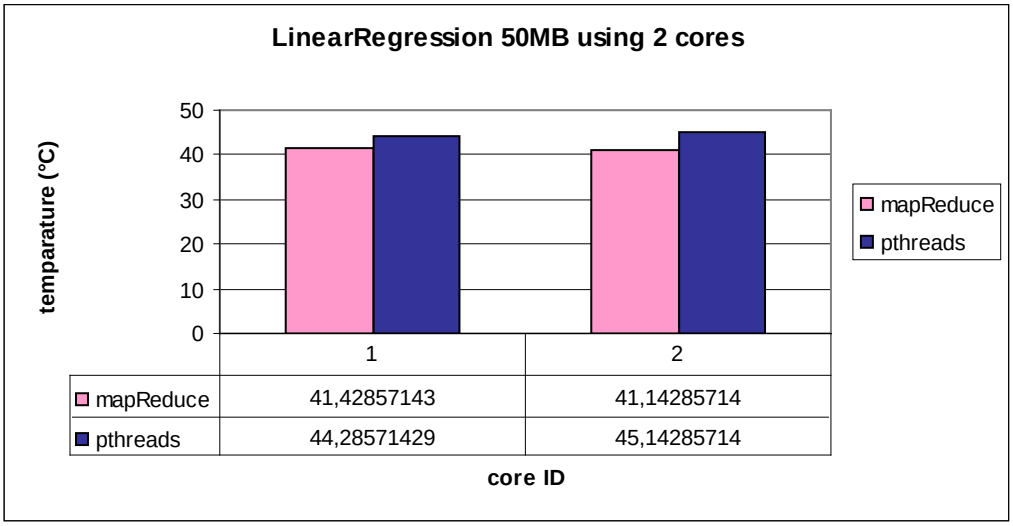
6.2.3 Linear Regression

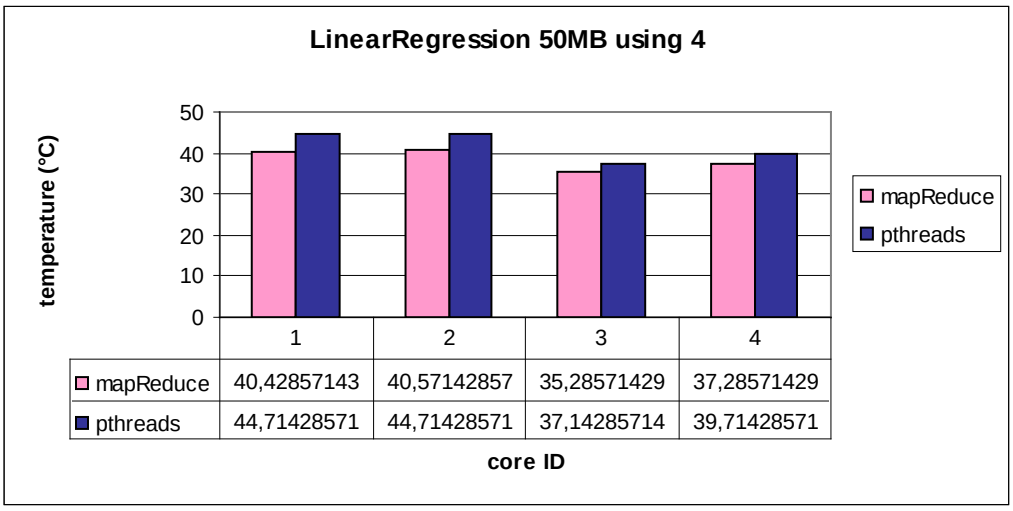
Για αυτή την εφαρμογή έτρεξα 4 σενάρια για 1, 2, 4 και 8 πυρήνες για κάθε αρχείο με τα δεδομένα εισόδου. Ουσιαστικά κάθε σενάριο έπρεπε να το τρέξω για το αρχείο μεγέθους 50MB, για το αρχείο μεγέθους 100MB, για το αρχείο μεγέθους 500MB και για το αρχείο μεγέθους 1GB.

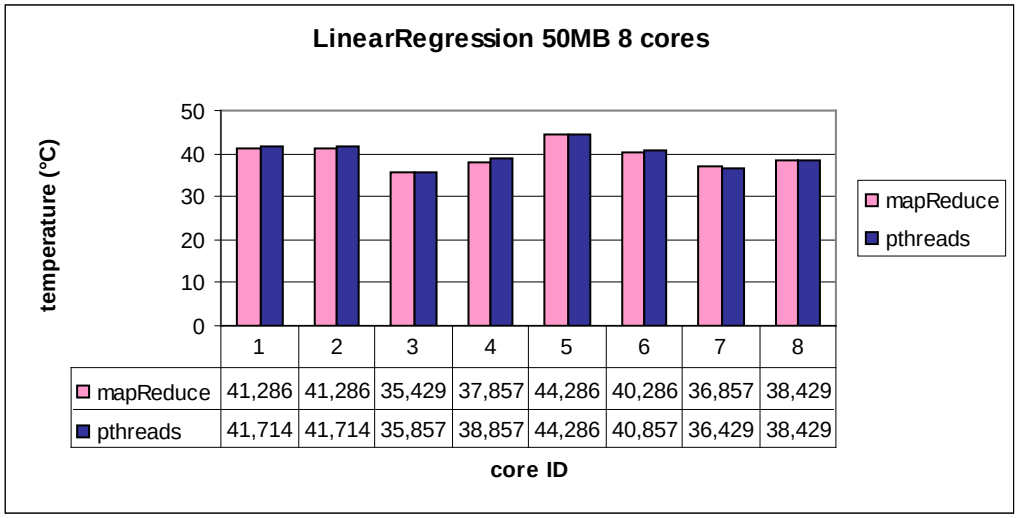
Linear Regression – Αρχείο μεγέθους 50MB

Στις πιο κάτω γραφικές (σχήμα 6.2.3.ι) παρουσιάζονται οι γραφικές για τα 4 σενάρια της εφαρμογής Linear Regression με αρχείο μεγέθους 50MB.


LinearRegression 50MB using 1 core
temperature (°C)
50
40
30
20
10
0
mapReduce
pthreads
core 1
mapReduce 42,85714286
pthreads 44,42857143
core ID


LinearRegression 50MB using 2 cores
temparature (°C)
50
40
30
20
10
0
mapReduce
pthreads
1
2
mapReduce 41,42857143 41,14285714
pthreads 44,28571429 45,14285714
core ID

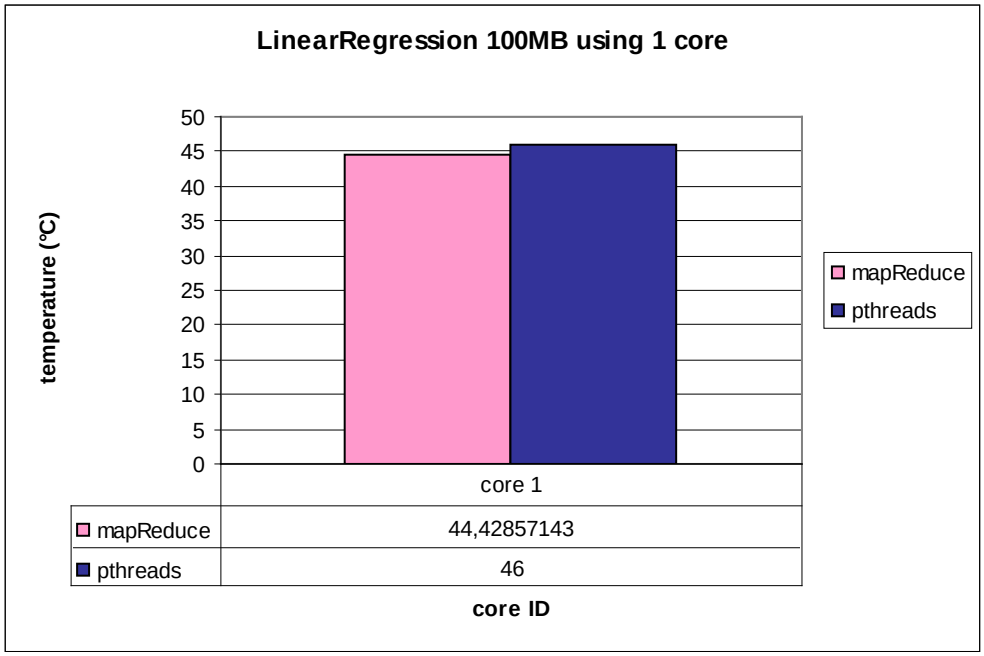

LinearRegression 50MB using 4
temperature (°C)
50
40
30
20
10
0
mapReduce
pthreads
1
2
3
4
mapReduce 40,42857143 40,57142857 35,28571429 37,28571429
pthreads 44,71428571 44,71428571 37,14285714 39,71428571
core ID

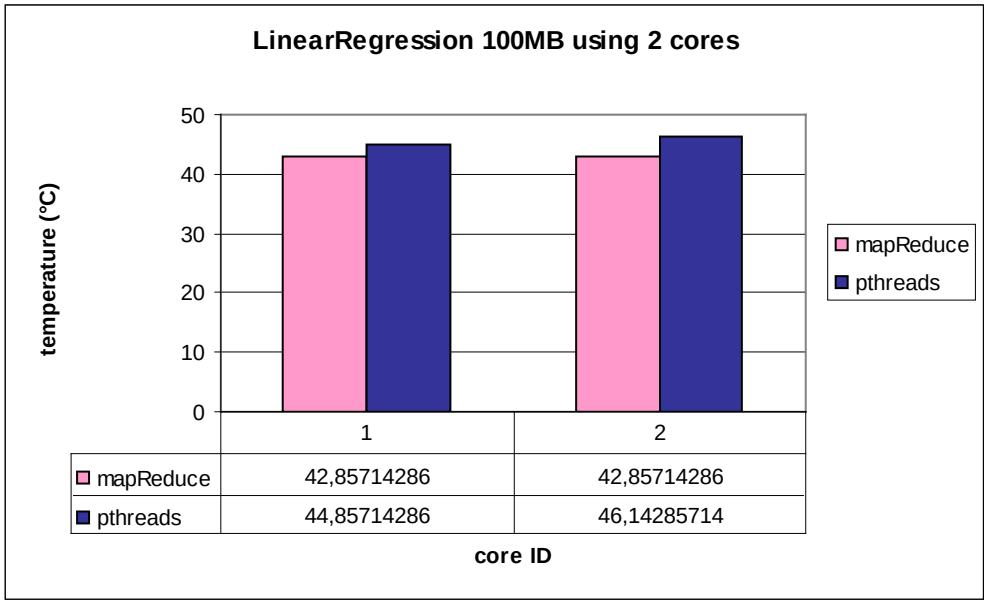


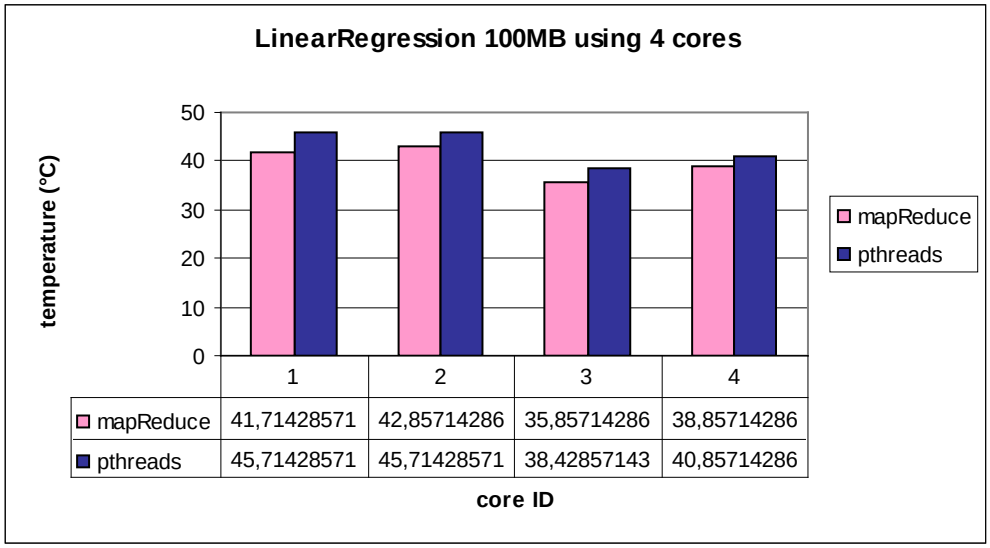
Σχήμα 6.2.3.ι: Γραφικές παραστάσεις εφαρμογής Linear Regression με αρχείο μεγέθους 50MB

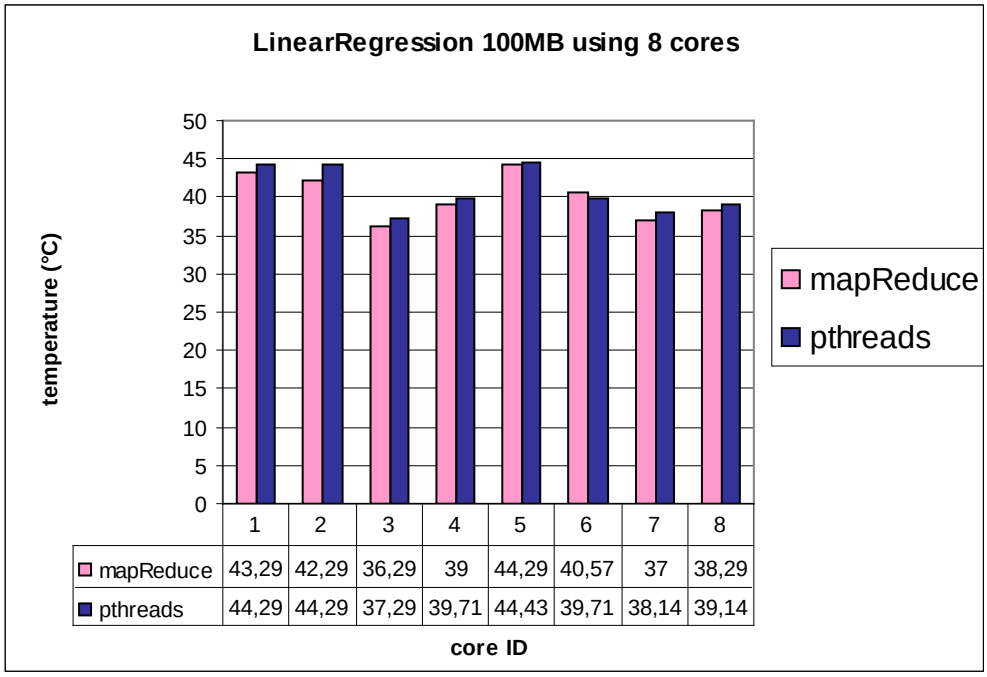
Linear Regression – Αρχείο μεγέθους 100MB

Στις πιο κάτω γραφικές (σχήμα 6.2.3.ιι) παρουσιάζονται οι γραφικές για τα 4 σενάρια της εφαρμογής Linear Regression με αρχείο μεγέθους 100MB.




LinearRegression 100MB using 2 cores
temperature (°C)
50
40
30
20
10
0
mapReduce
pthreads
1
2
mapReduce 42,85714286 42,85714286
pthreads 44,85714286 46,14285714
core ID

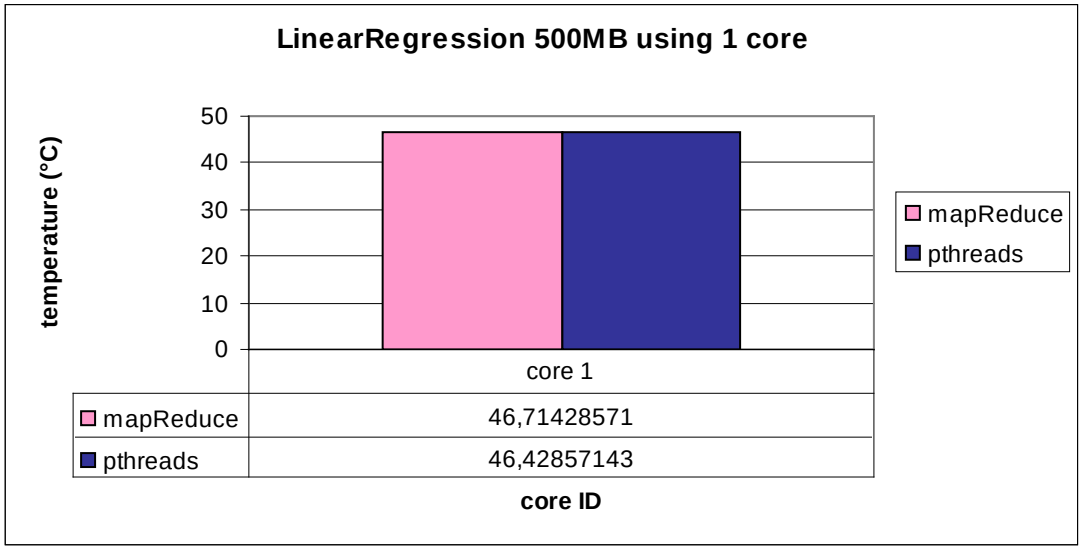

LinearRegression 100MB using 4 cores
temperature (°C)
50
40
30
20
10
0
mapReduce
pthreads
1
2
3
4
mapReduce 41,71428571 42,85714286 35,85714286 38,85714286
pthreads 45,71428571 45,71428571 38,42857143 40,85714286
core ID

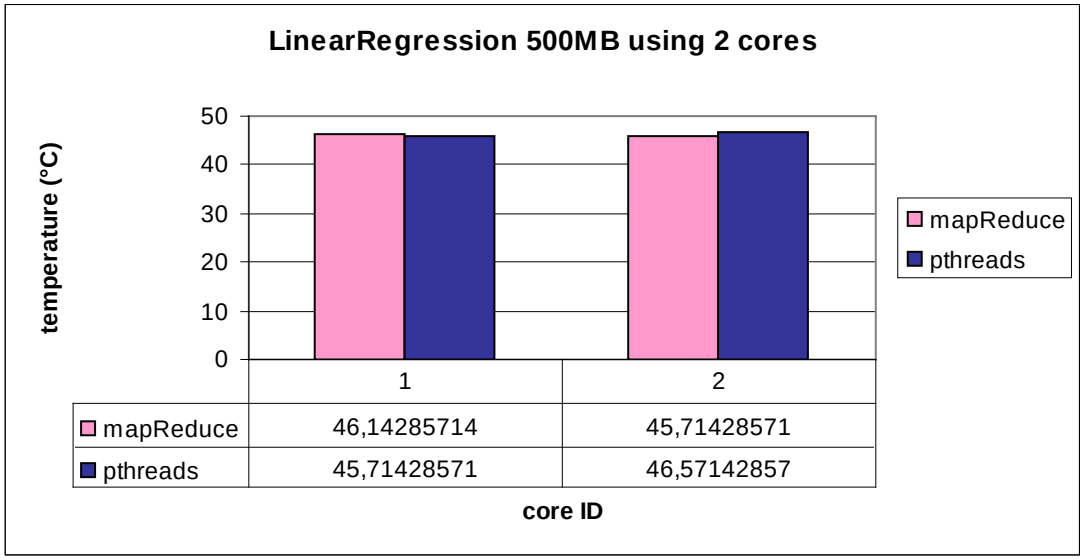


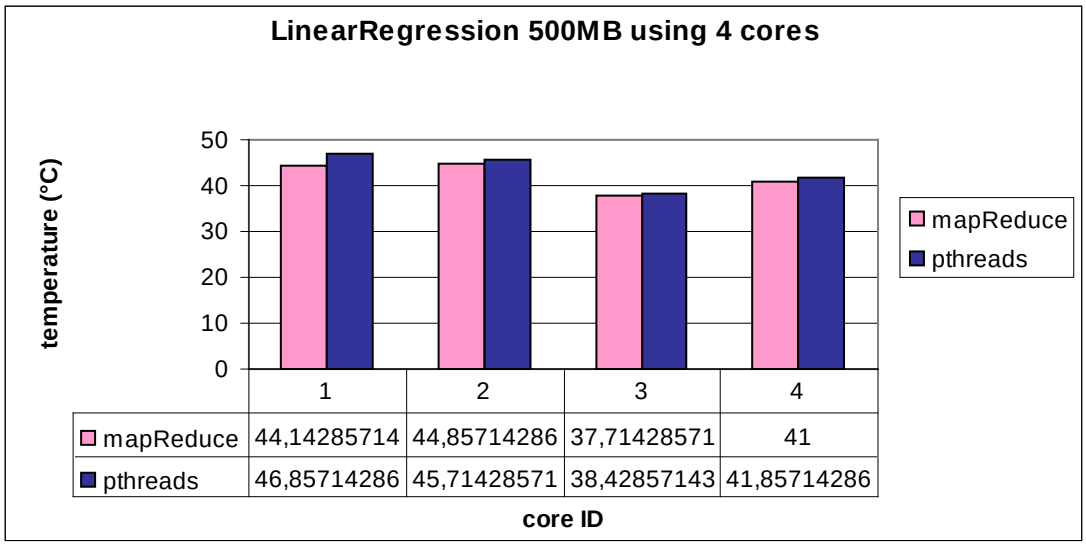
Σχήμα 6.2.3.ιι: Γραφικές παραστάσεις εφαρμογής Linear Regression με αρχείο μεγέθους 100MB

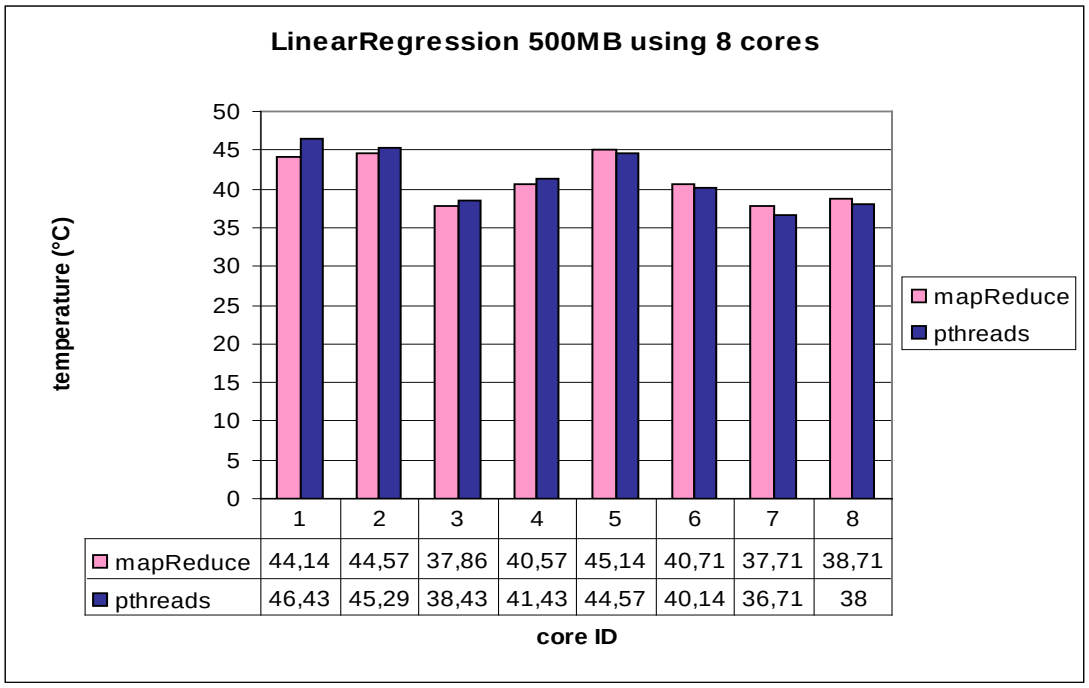
Linear Regression – Αρχείο μεγέθους 500MB

Στις πιο κάτω γραφικές (σχήμα 6.2.3.ιιι) παρουσιάζονται οι γραφικές για τα 4 σενάρια της εφαρμογής Linear Regression με αρχείο μεγέθους 500MB.




LinearRegression 500MB using 2 cores
temperature (°C)
50
40
30
20
10
0
mapReduce
pthreads
1
2
mapReduce 46,14285714 45,71428571
pthreads 45,71428571 46,57142857
core ID

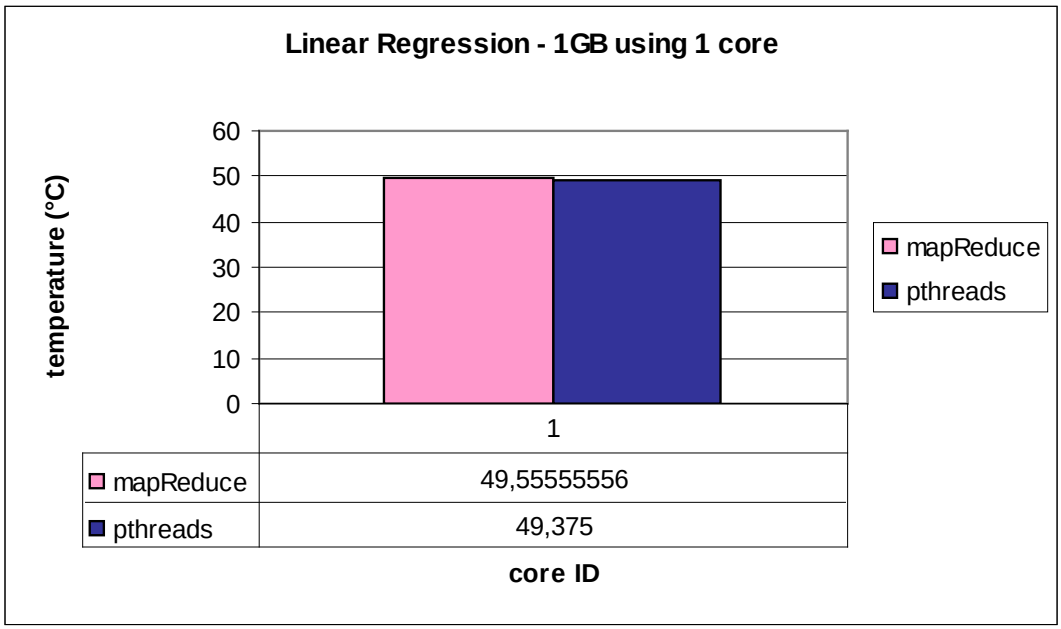

LinearRegression 500MB using 4 cores
temperature (°C)
50
40
30
20
10
0
mapReduce
pthreads
1
2
3
4
mapReduce 44,14285714 44,85714286 37,71428571 41
pthreads 46,85714286 45,71428571 38,42857143 41,85714286
core ID

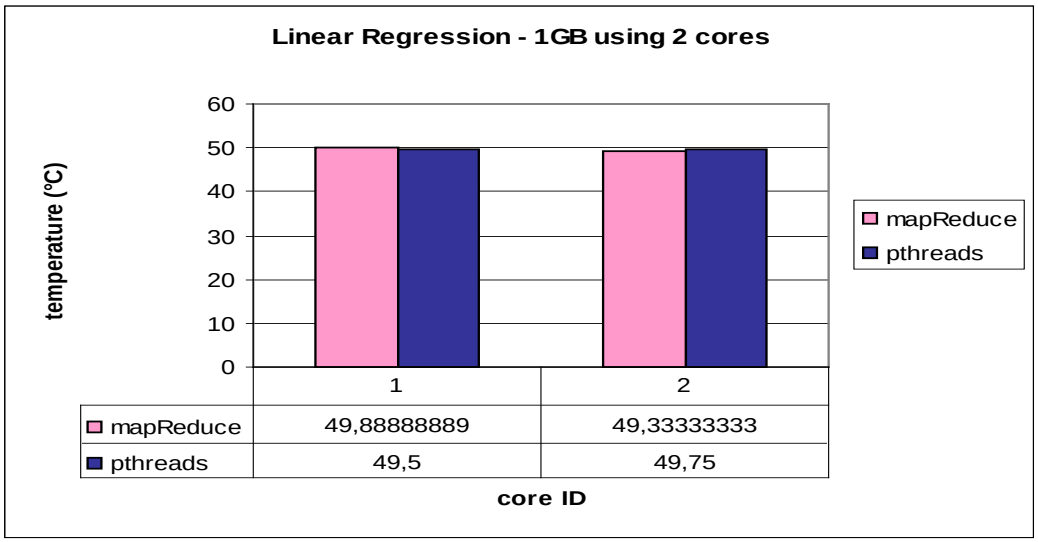


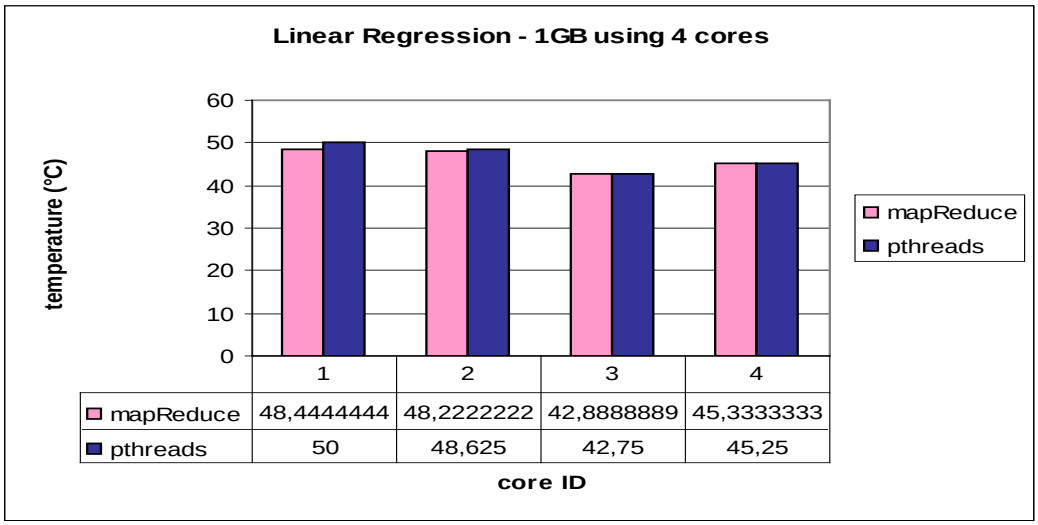
Σχήμα 6.2.3.iii: Γραφικές παραστάσεις εφαρμογής Linear Regression με αρχείο μεγέθους 500MB

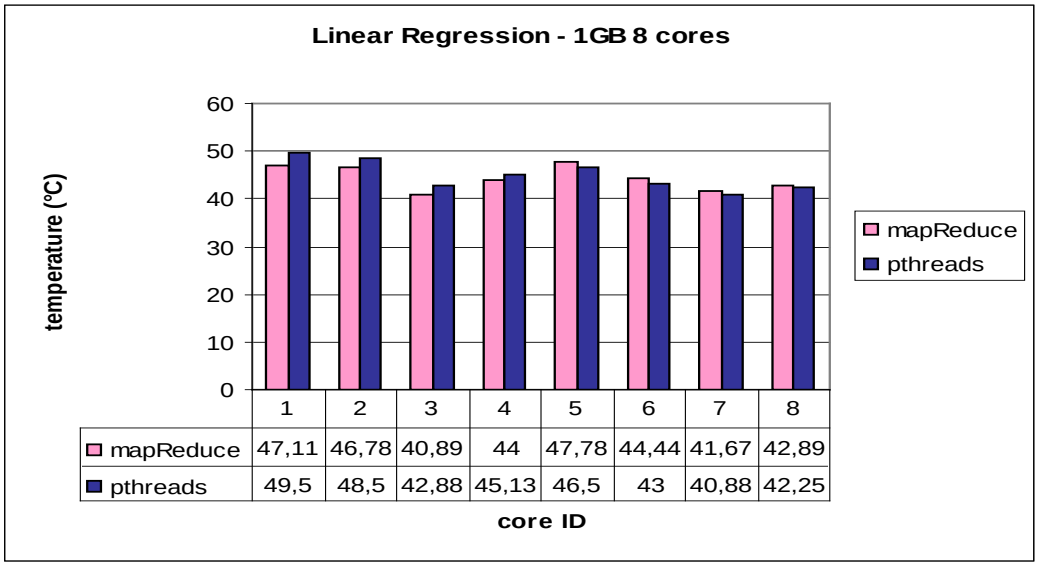
Linear Regression – Αρχείο μεγέθους 1GB

Στις πιο κάτω γραφικές (σχήμα 6.2.3.iiii) παρουσιάζονται οι γραφικές για τα 4 σενάρια της εφαρμογής Linear Regression με αρχείο μεγέθους 1GB.









Σχήμα 6.2.3.iιι: Γραφικές παραστάσεις εφαρμογής Linear Regression με αρχείο μεγέθους 1GB

Στα σχήματα 6.2.3.ι, 6.2.3.ιι, 6.2.3.ιιι και 6.2.3.ιιιι παρουσιάζεται ο μέσος όρος των τιμών στην θερμοκρασία κάθε πυρήνα όπως έχουν καταγραφεί από την εκτέλεση της εφαρμογής Linear Regression. Τα αποτελέσματα είναι σε σύγκριση με τα αντίστοιχα αποτελέσματα της ίδιας εφαρμογής υλοποιημένης σε pthreads.

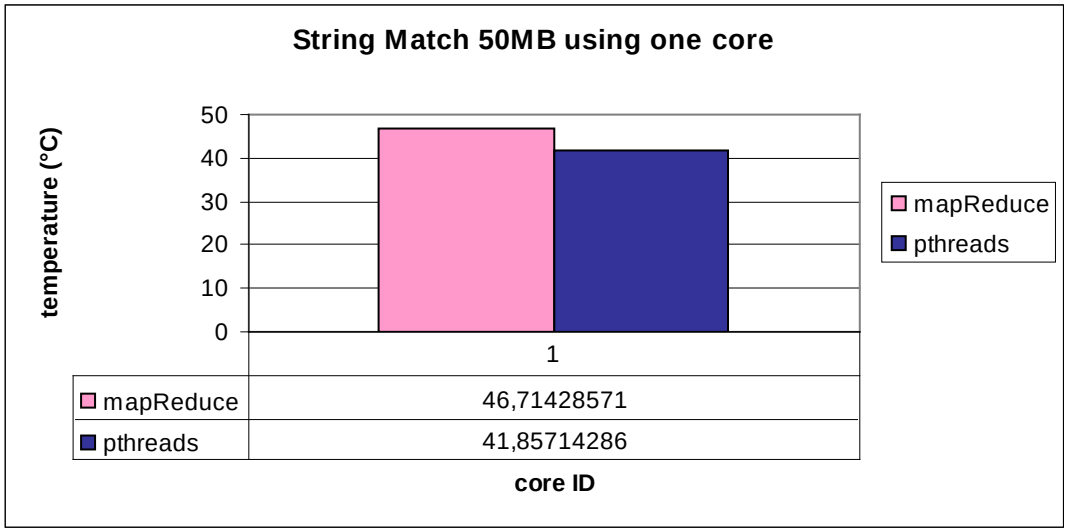
Το ενδιαφέρον εδώ παρουσιάζεται στο γεγονός ότι η ίδια εφαρμογή έτρεξε για κάθε περίπτωση με 3 διαφορετικά αρχεία εισόδου διαφορετικού μεγέθους. Και πάλι οι τιμές που καταγράφηκαν δεν είχαν σημαντικές διαφορές ανάμεσα στις δύο υλοποιήσεις. Επίσης, όταν η εφαρμογή τρέχει για την ίδια υλοποίηση με διαφορετικό αρχείο εισόδου παρατηρείται μερική αύξηση στην τιμή της θερμοκρασίας των πυρήνων όταν είναι μεγαλύτερο το αρχείο. Αυτό οφείλεται στο μεγαλύτερο χρόνο που χρειάζεται η εφαρμογή για να τρέξει για μεγαλύτερα αρχεία, χωρίς όμως να παρατηρείται μεγάλη αύξηση στις τιμές της θερμοκρασίας.

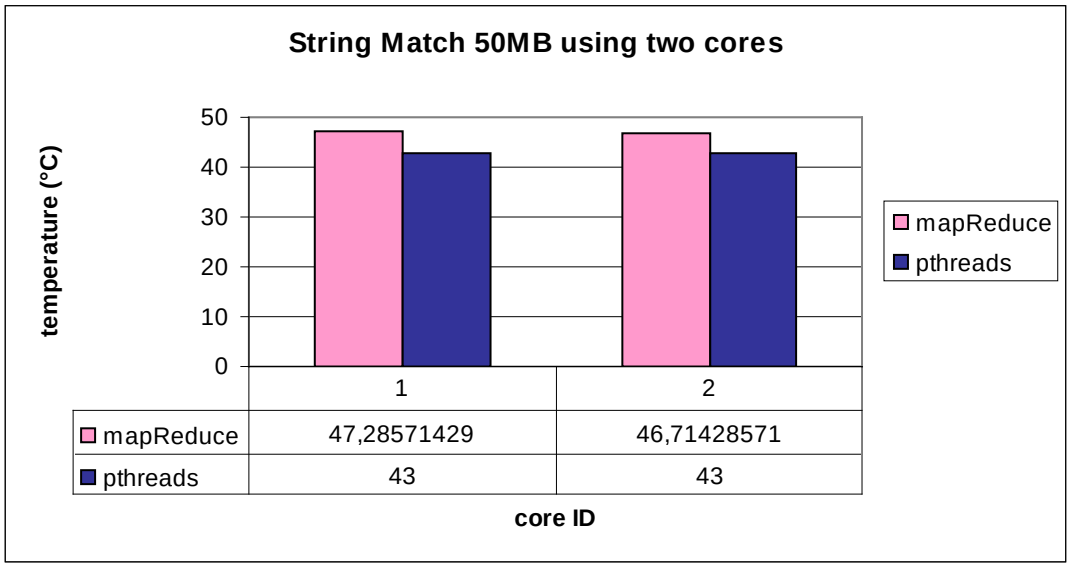
6.2.4 String Match

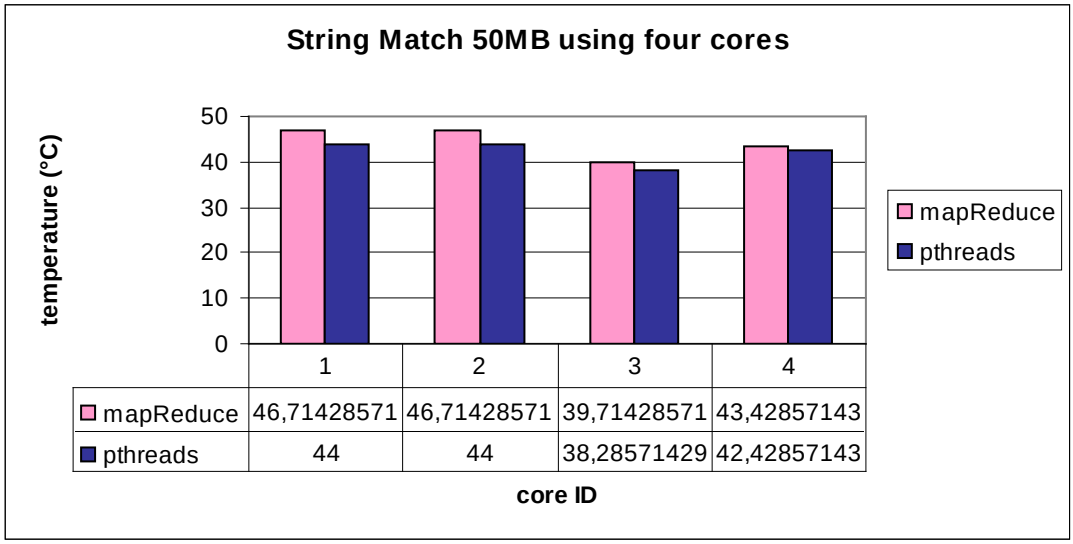
Για αυτή την εφαρμογή έτρεξα 4 σενάρια για 1, 2, 4 και 8 πυρήνες για κάθε αρχείο με τα δεδομένα εισόδου. Ουσιαστικά κάθε σενάριο έπρεπε να το τρέξω για το αρχείο μεγέθους 50MB, για το αρχείο μεγέθους 100MB, για το αρχείο μεγέθους 500MB και για το αρχείο μεγέθους 1GB.

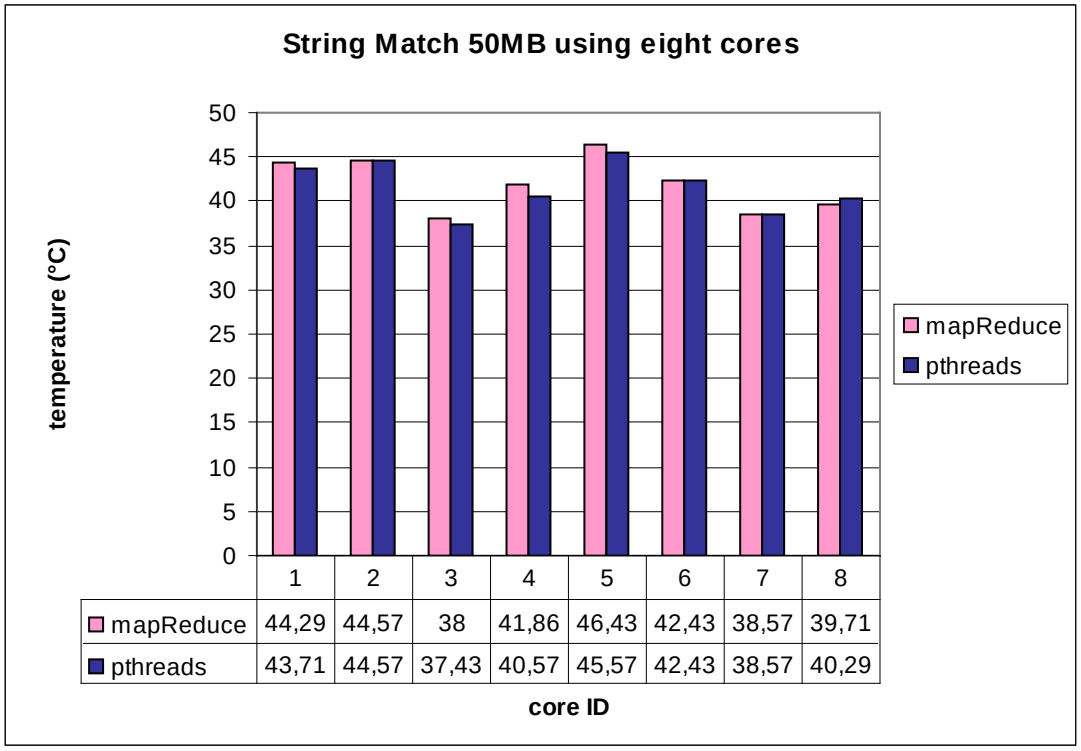
String Match – Αρχείο μεγέθους 50MB

Στις πιο κάτω γραφικές (σχήμα 6.2.4.ι) παρουσιάζονται οι γραφικές για τα 4 σενάρια της εφαρμογής String Match με αρχείο μεγέθους 50MB.


String Match 50MB using one core
temperature (°C)
50
40
30
20
10
0
1
mapReduce
pthreads
mapReduce 46,71428571
pthreads 41,85714286
core ID


String Match 50MB using two cores
temperature (°C)
50
40
30
20
10
0
1
2
mapReduce
pthreads
mapReduce 47,28571429 46,71428571
pthreads 43 43
core ID


String Match 50MB using four cores
temperature (°C)
50
40
30
20
10
0
1
2
3
4
mapReduce
pthreads
mapReduce 46,71428571 46,71428571 39,71428571 43,42857143
pthreads 44 44 38,28571429 42,42857143
core ID

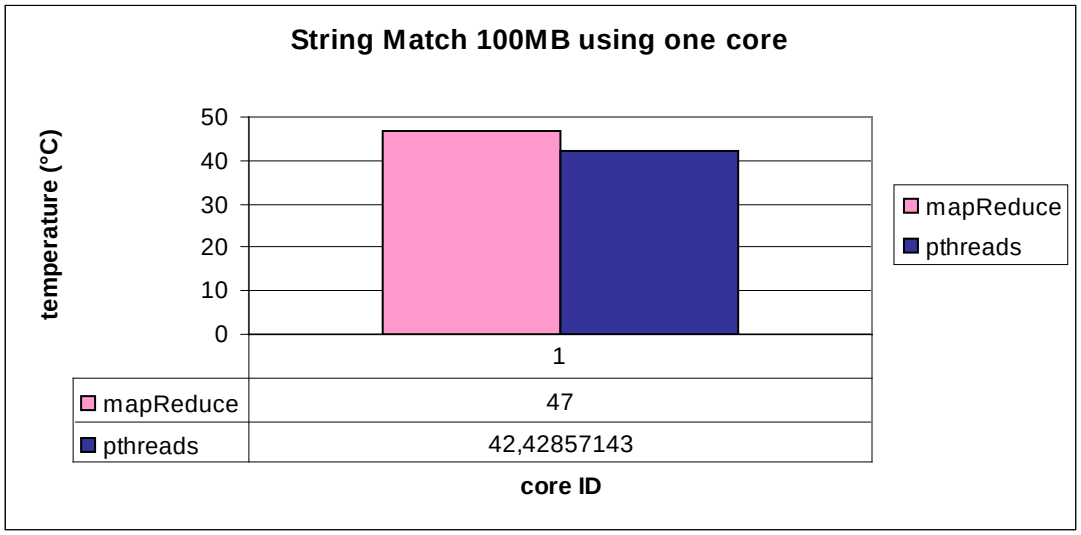


	1	2	3	4	5	6	7	8
mapReduce	44,29	44,57	38	41,86	46,43	42,43	38,57	39,71
pthreads	43,71	44,57	37,43	40,57	45,57	42,43	38,57	40,29

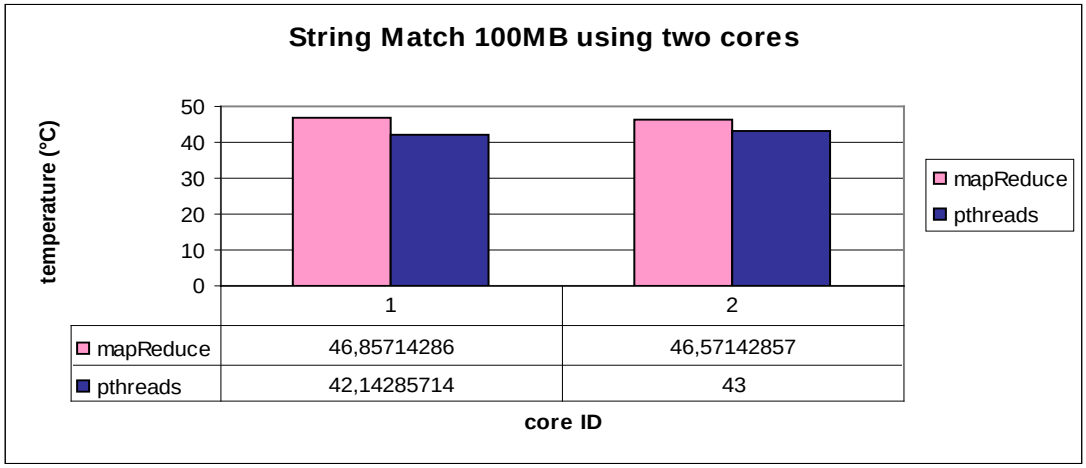
Σχήμα 6.2.4.ι: Γραφικές παραστάσεις εφαρμογής String Match με αρχείο μεγέθους 50MB

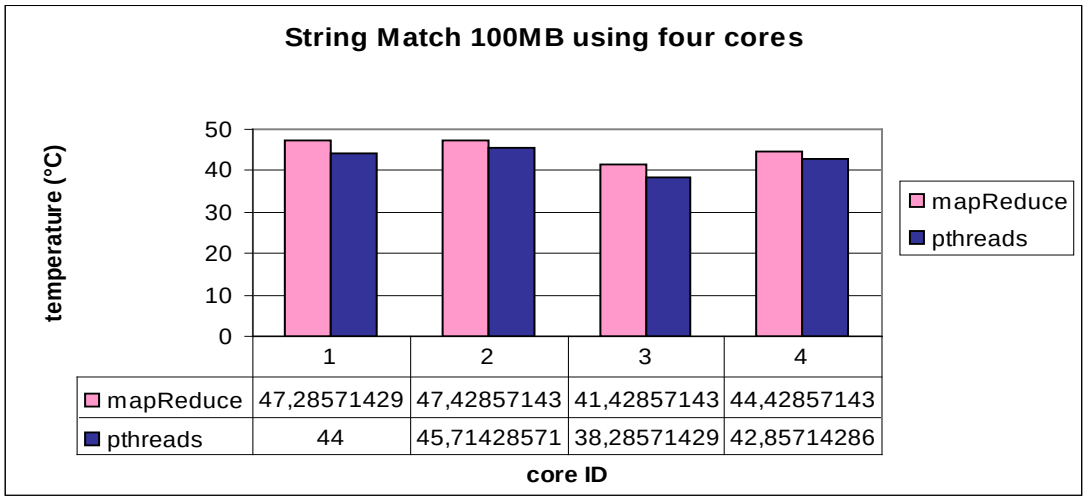
String Match – Αρχείο μεγέθους 100MB

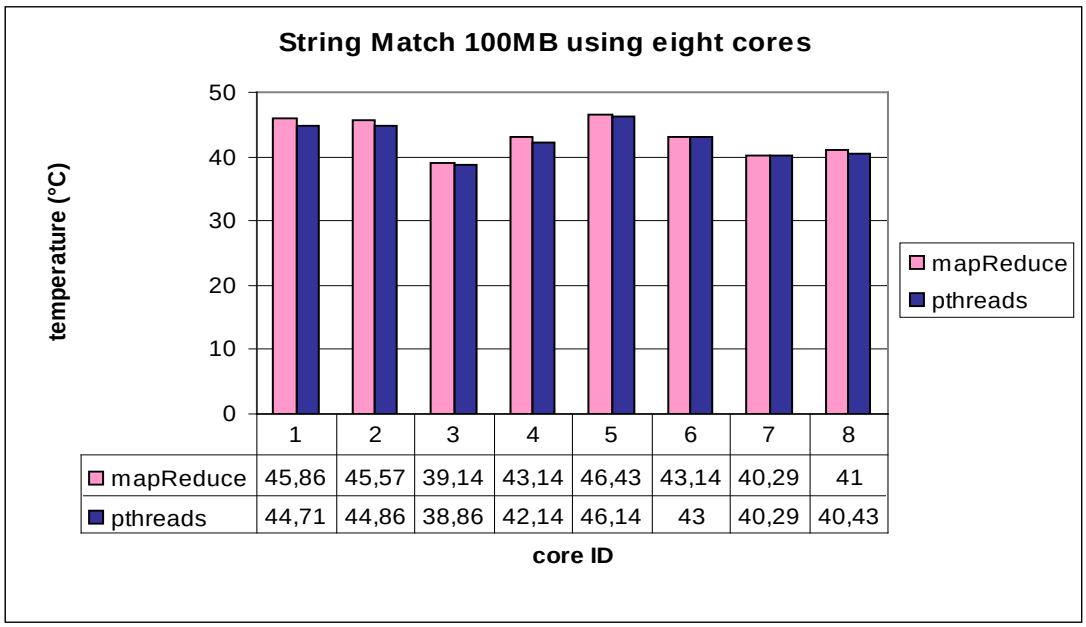
Στις πιο κάτω γραφικές (σχήμα 6.2.4.ιι) παρουσιάζονται οι γραφικές για τα 4 σενάρια της εφαρμογής String Match με αρχείο μεγέθους 100MB.



	1
mapReduce	47
pthreads	42,42857143



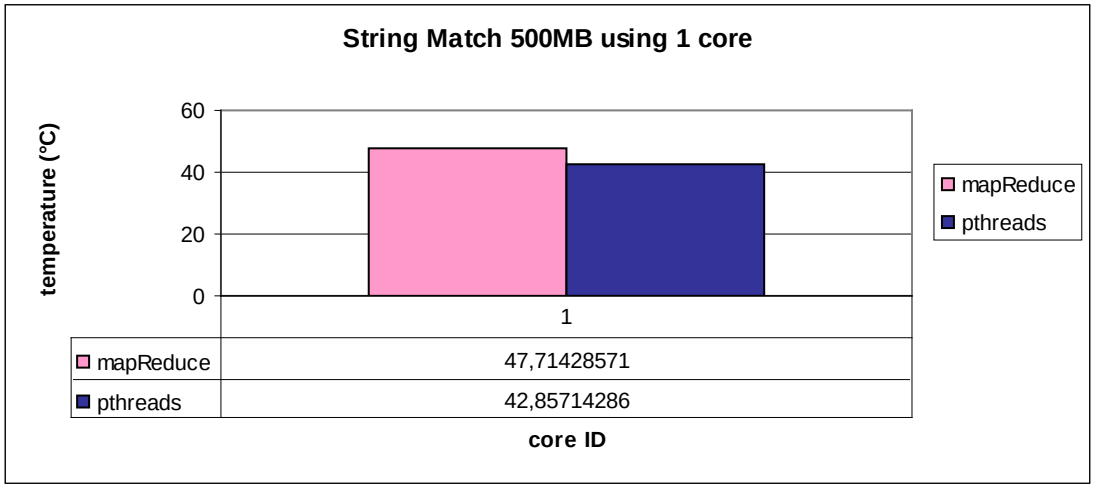


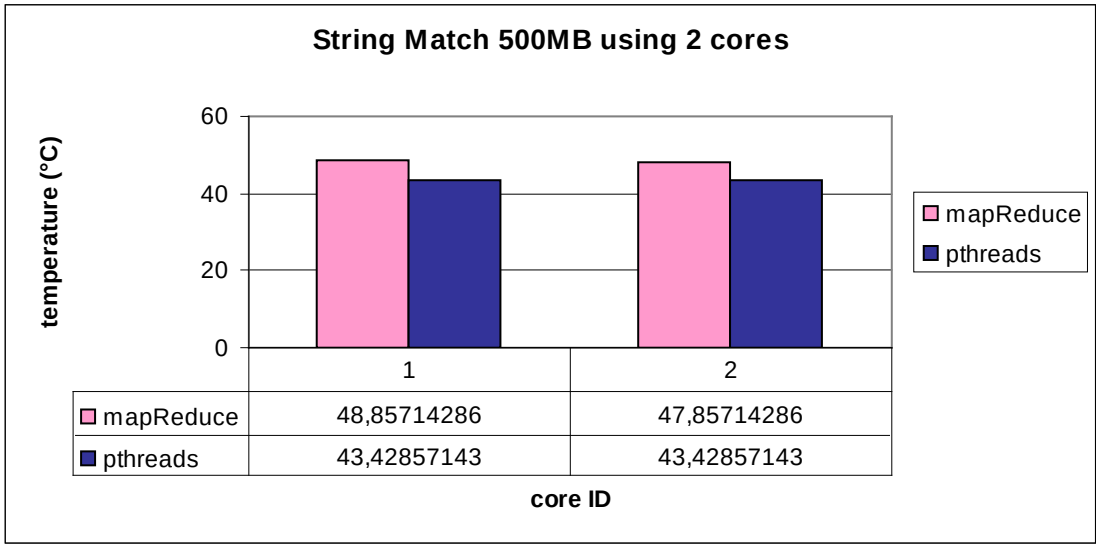


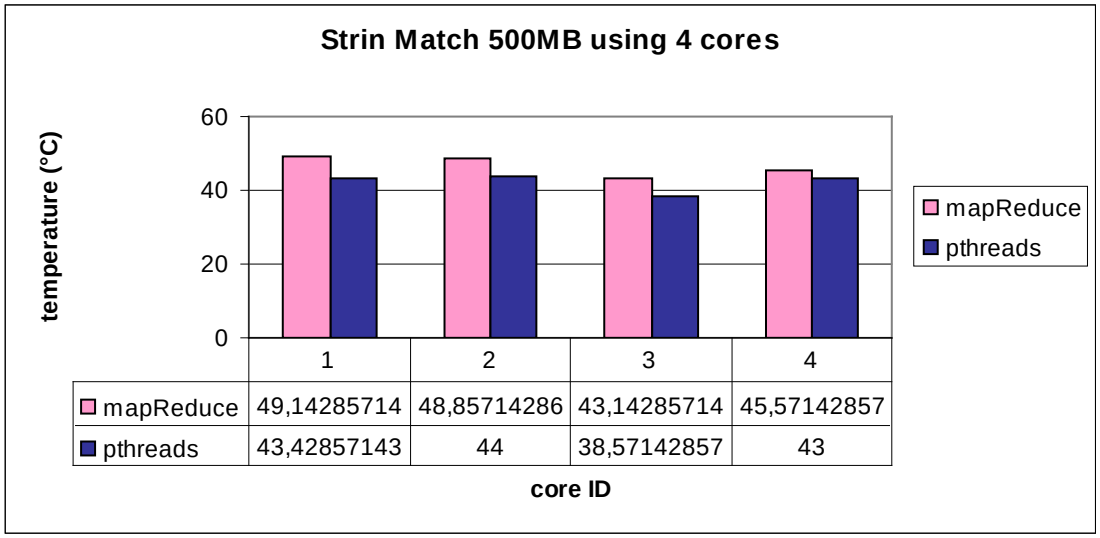
Σχήμα 6.2.4.ιι: Γραφικές παραστάσεις εφαρμογής String Match με αρχείο μεγέθους 100MB

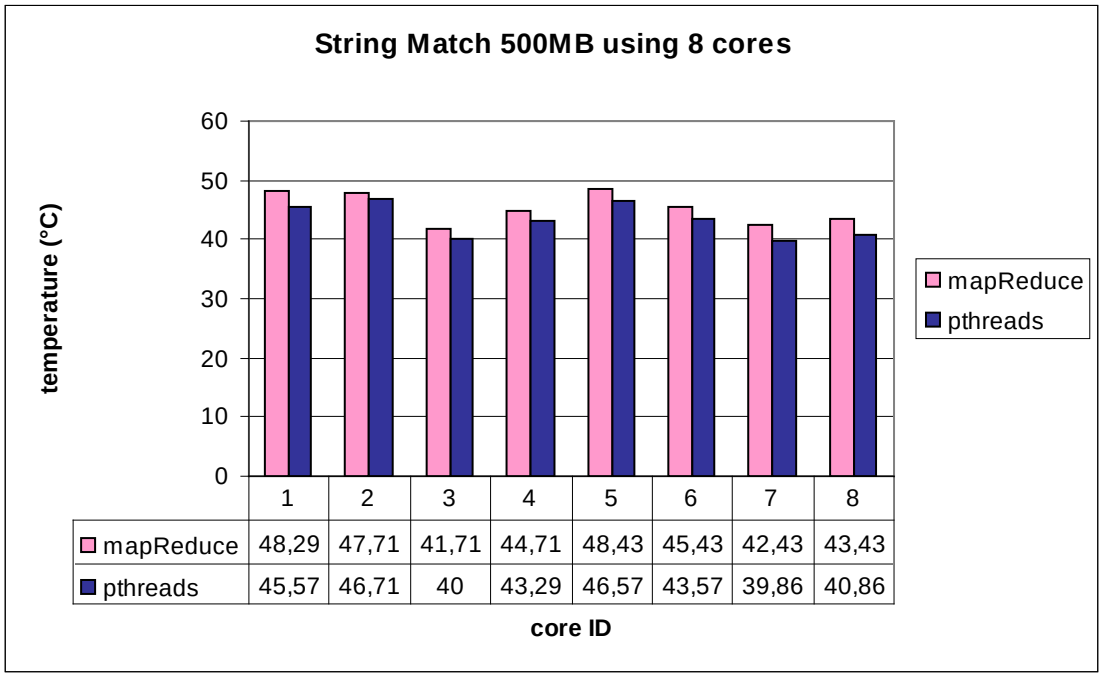
String Match – Αρχείο μεγέθους 500MB

Στις πιο κάτω γραφικές (σχήμα 6.2.4.ιιι) παρουσιάζονται οι γραφικές για τα 4 σενάρια της εφαρμογής String Match με αρχείο μεγέθους 500MB.





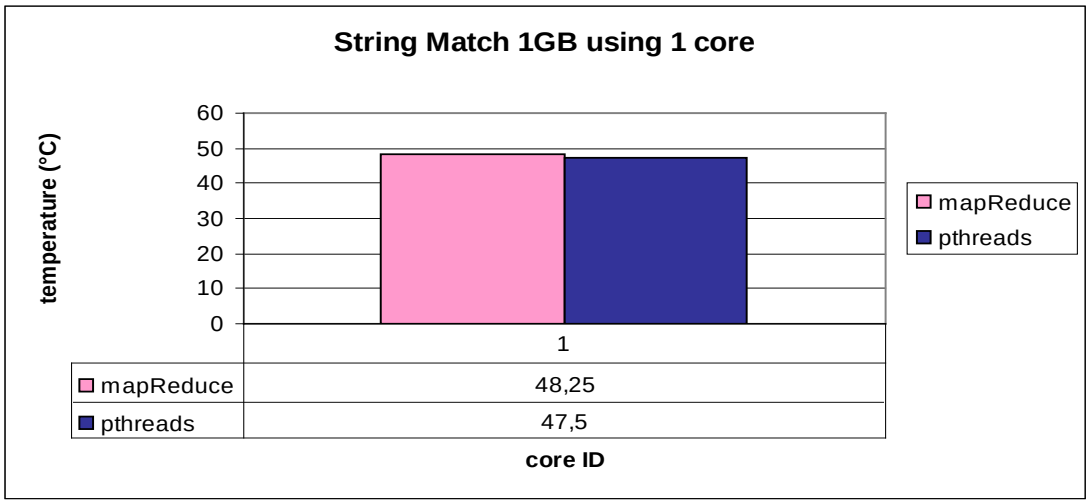


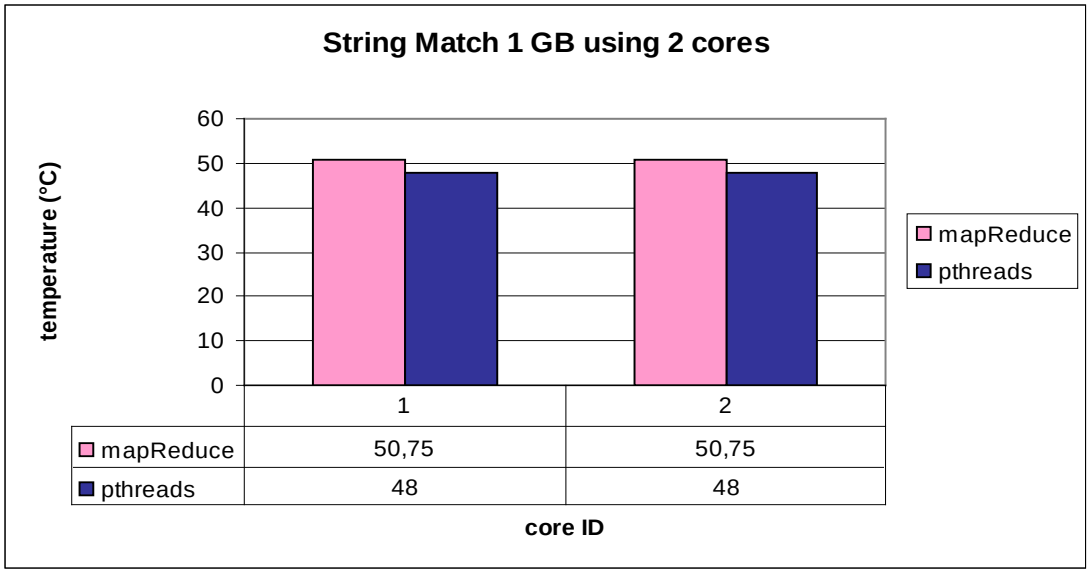


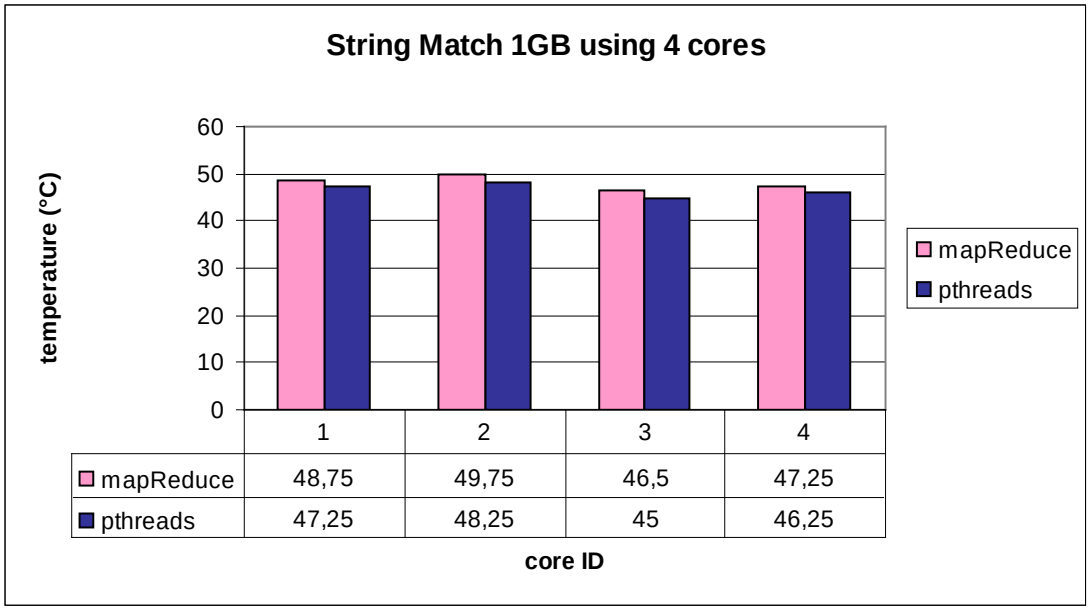
Σχήμα 6.2.4.ιιι: Γραφικές παραστάσεις εφαρμογής String Match με αρχείο μεγέθους 500MB

String Match – Αρχείο μεγέθους 1GB

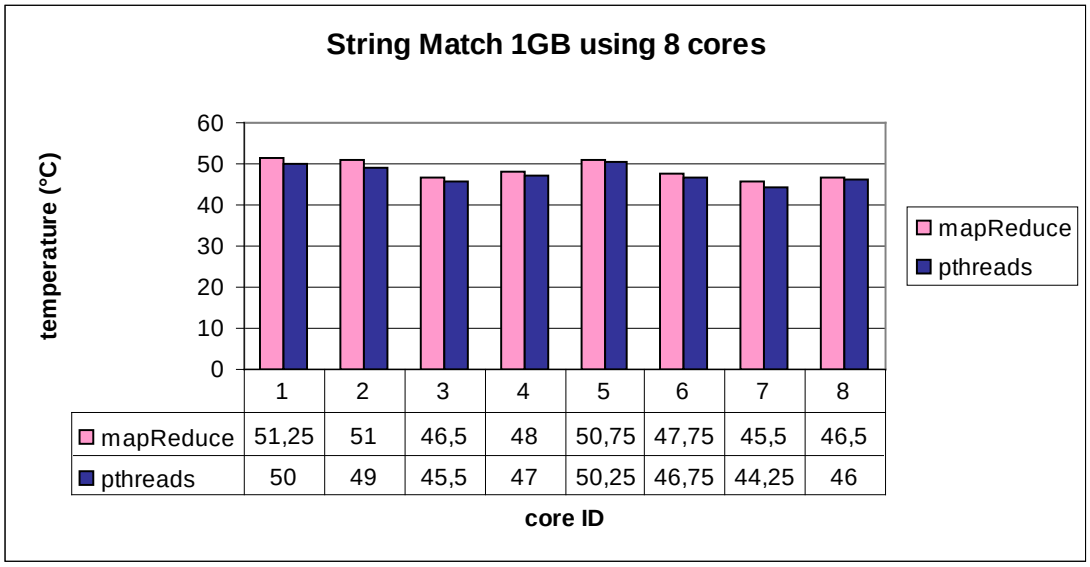
Στις πιο κάτω γραφικές (σχήμα 6.2.4.ιιι) παρουσιάζονται οι γραφικές για τα 4 σενάρια της εφαρμογής String Match με αρχείο μεγέθους 1ΓΒ.







	1	2	3	4
mapReduce	48,75	49,75	46,5	47,25
pthreads	47,25	48,25	45	46,25



	1	2	3	4	5	6	7	8
mapReduce	51,25	51	46,5	48	50,75	47,75	45,5	46,5
pthreads	50	49	45,5	47	50,25	46,75	44,25	46

Σχήμα 6.2.4.ιιι: Γραφικές παραστάσεις εφαρμογής String Match με αρχείο μεγέθους 1GB

Στα σχήματα 6.2.4.ι, 6.2.4.ιι, 6.2.4.ιιι και 6.2.4.ιιιι παρουσιάζεται ο μέσος όρος των τιμών στην θερμοκρασία κάθε πυρήνα όπως έχουν καταγραφεί από την εκτέλεση της εφαρμογής String Match. Τα αποτελέσματα είναι σε σύγκριση με τα αντίστοιχα αποτελέσματα της ίδιας εφαρμογής υλοποιημένης σε pthreads.

Στην περίπτωση της εφαρμογής string match η οποία έχει εκτελεστεί για κάθε υλοποίηση με τέσσερα διαφορετικά αρχεία εισόδου διαφορετικού μεγέθους δεν

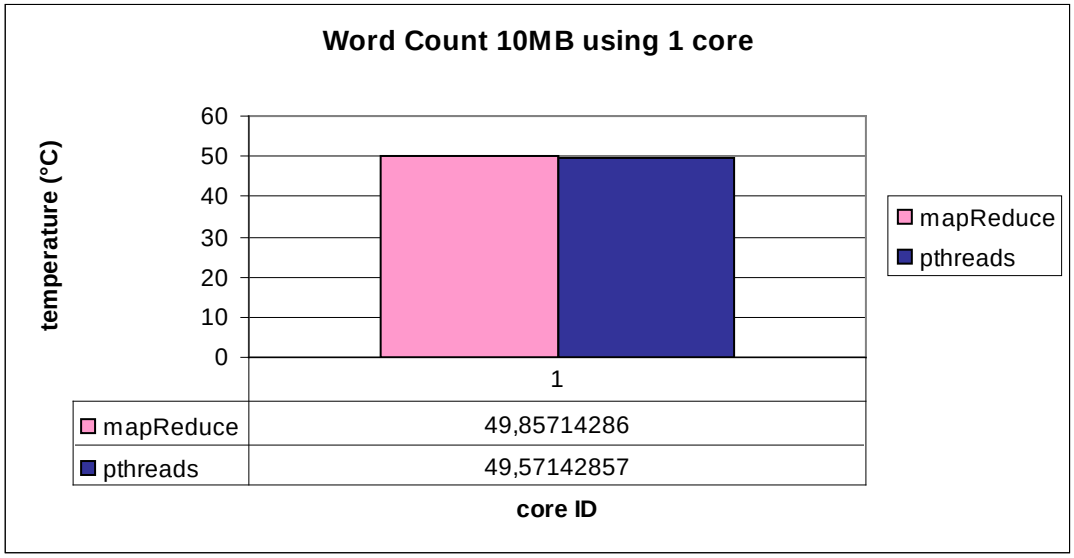
παρουσιάζονται και πάλι διαφορές ανάμεσα στις δύο διαφορετικές υλοποιήσεις. Συγκρίνοντας και πάλι την υλοποίηση του Phoenix όταν τρέχει με τον ίδιο αριθμό πυρήνων για διαφορετικό μέγεθος αρχείου εισόδου, δεν παρατηρούμε οποιαδήποτε σημαντική διαφορά στη θερμοκρασία.

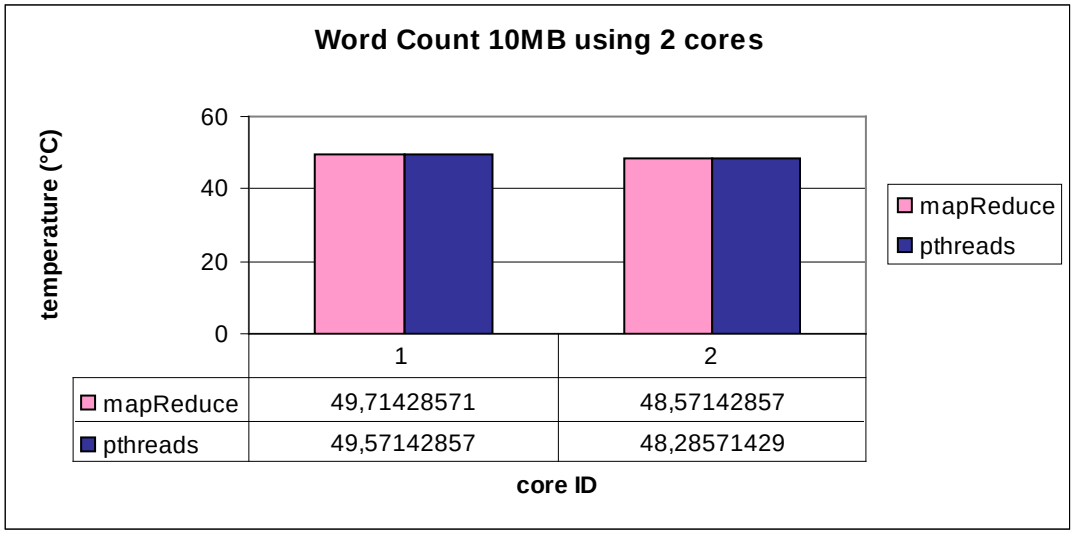
6.2.5 Word Count

Για αυτή την εφαρμογή έτρεξα 4 σενάρια για 1, 2, 4 και 8 πυρήνες για κάθε αρχείο με τα δεδομένα εισόδου. Ουσιαστικά κάθε σενάριο έπρεπε να το τρέξω για το αρχείο μεγέθους 10MB, για το αρχείο μεγέθους 50MB, για το αρχείο μεγέθους 100MB, για το αρχείο μεγέθους 500MB και για το αρχείο μεγέθους 1GB.

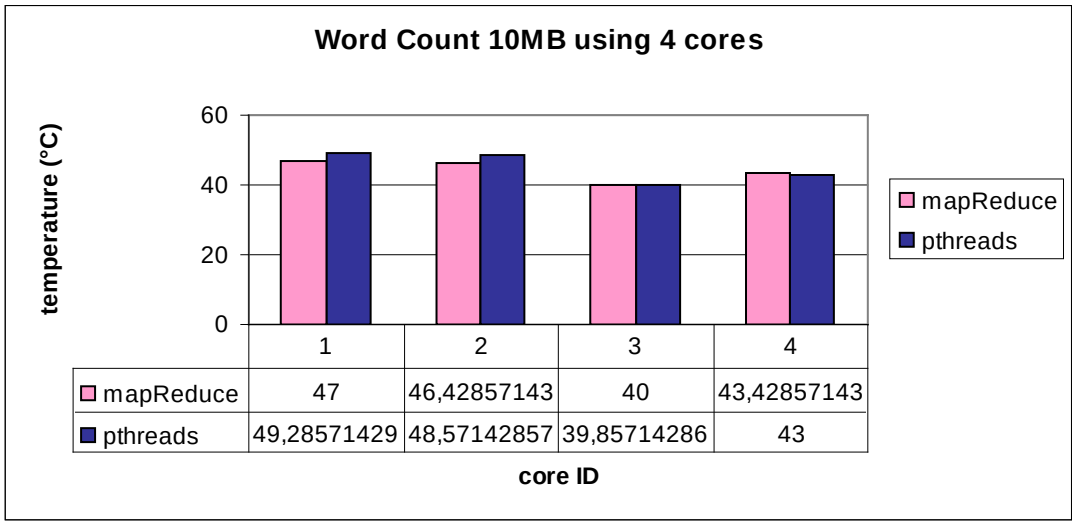
Word Count – Αρχείο μεγέθους 10MB

Στις πιο κάτω γραφικές (σχήμα 6.2.5.ι) παρουσιάζονται οι γραφικές για τα 4 σενάρια της εφαρμογής Word Count με αρχείο μεγέθους 10MB.

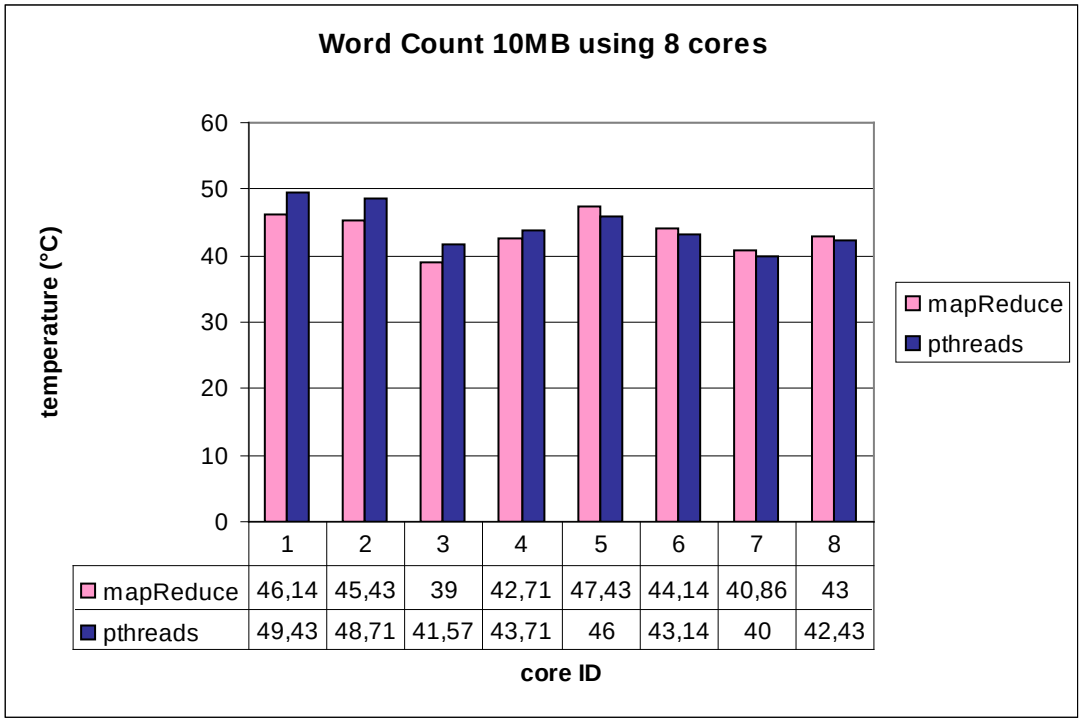




	1	2
mapReduce	49,71428571	48,57142857
pthreads	49,57142857	48,28571429



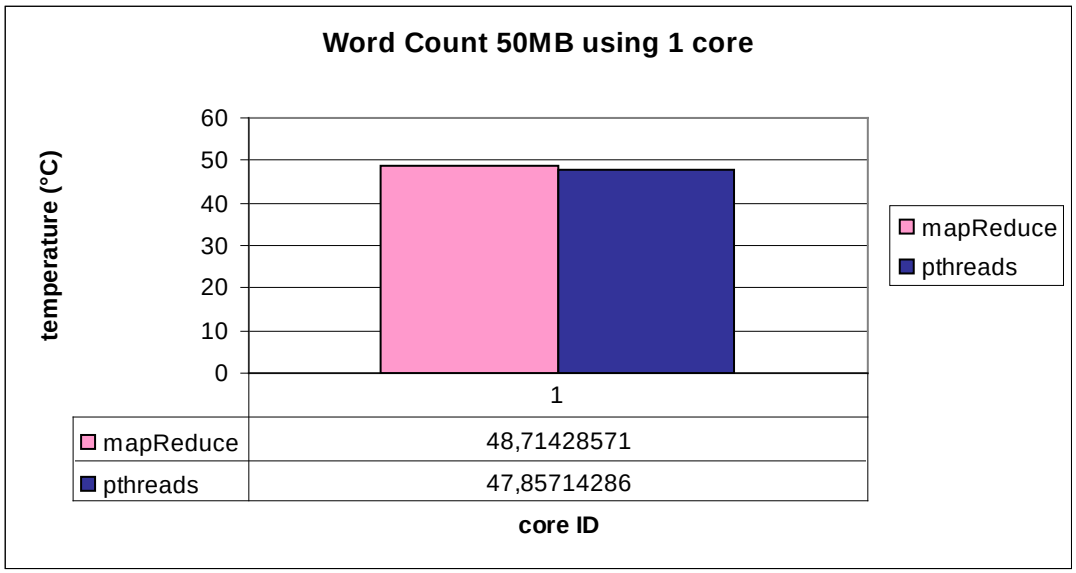
	1	2	3	4
mapReduce	47	46,42857143	40	43,42857143
pthreads	49,28571429	48,57142857	39,85714286	43

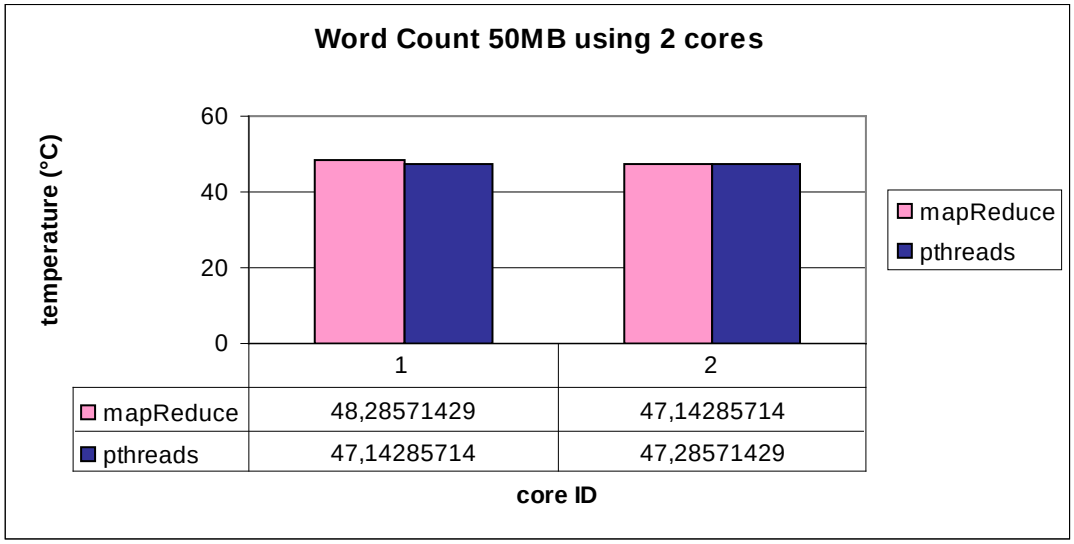


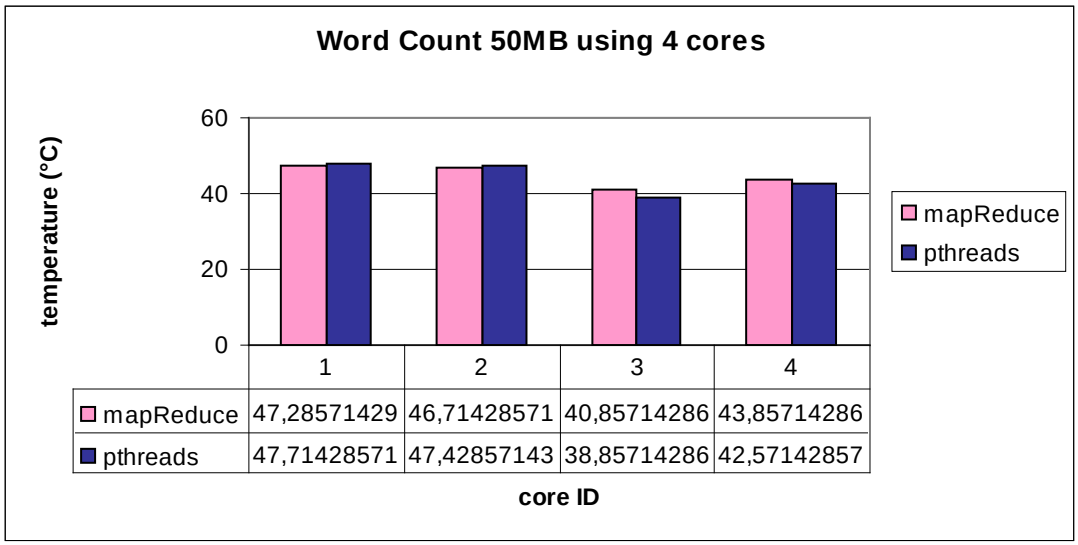
Σχήμα 6.2.5.ι: Γραφικές παραστάσεις εφαρμογής Word Count με αρχείο μεγέθους 10MB

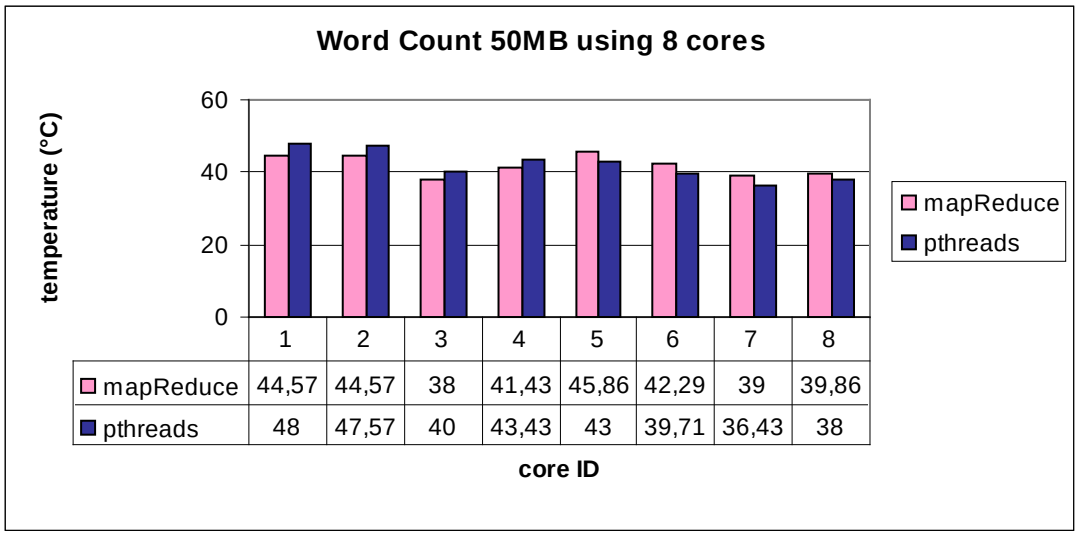
Word Count – Αρχείο μεγέθους 50MB

Στις πιο κάτω γραφικές (σχήμα 6.2.5.ιι) παρουσιάζονται οι γραφικές για τα 4 σενάρια της εφαρμογής Word Count με αρχείο μεγέθους 50MB.





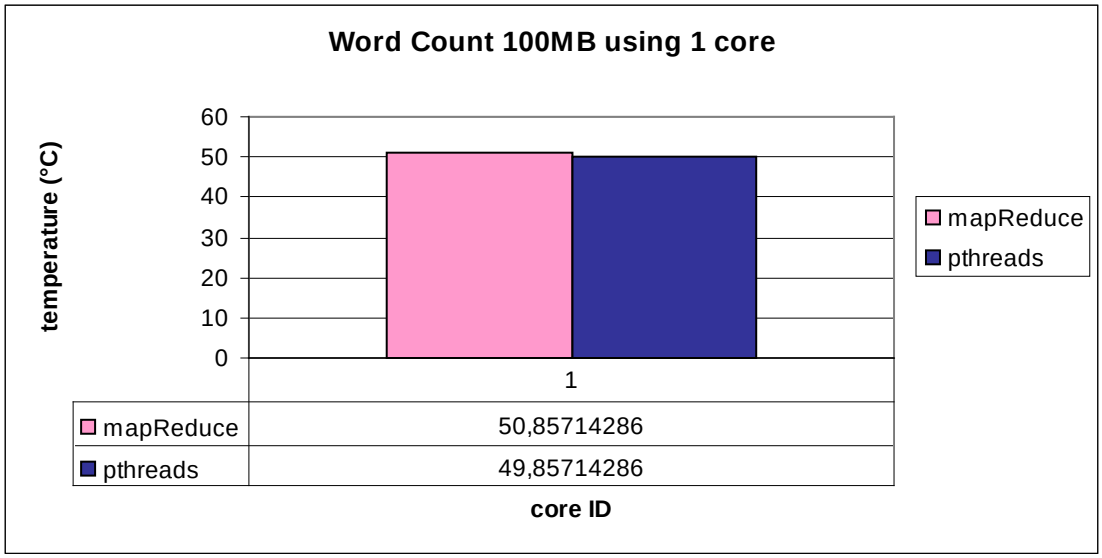


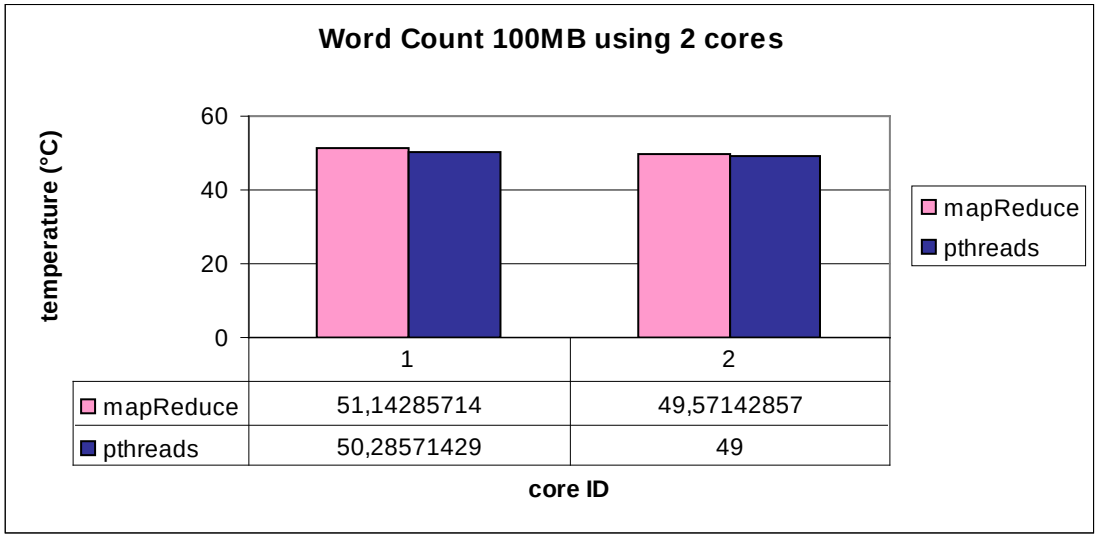


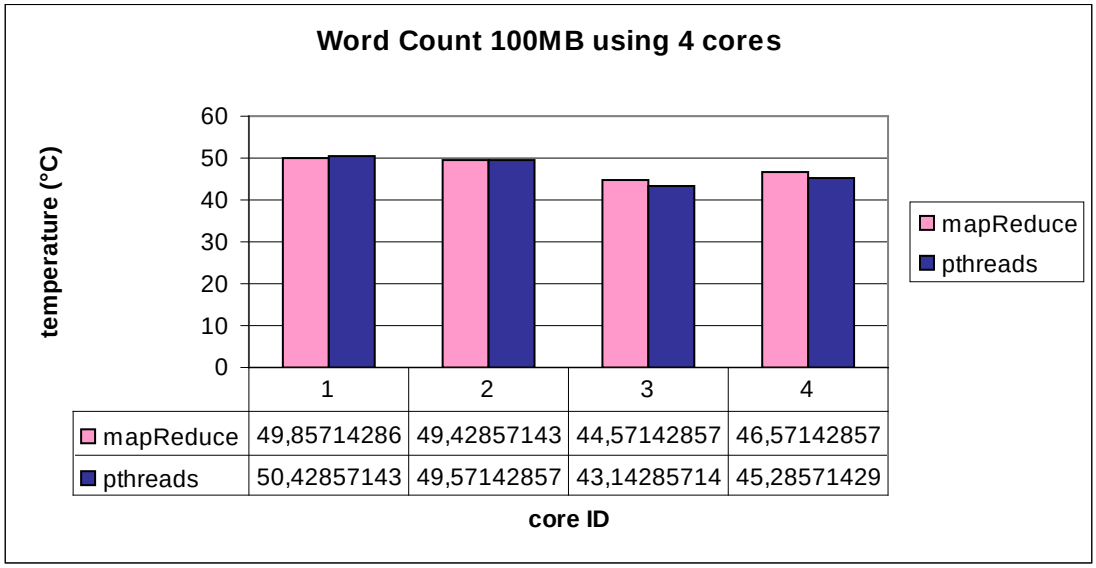
Σχήμα 6.2.5.ιι: Γραφικές παραστάσεις εφαρμογής Word Count με αρχείο μεγέθους 50MB

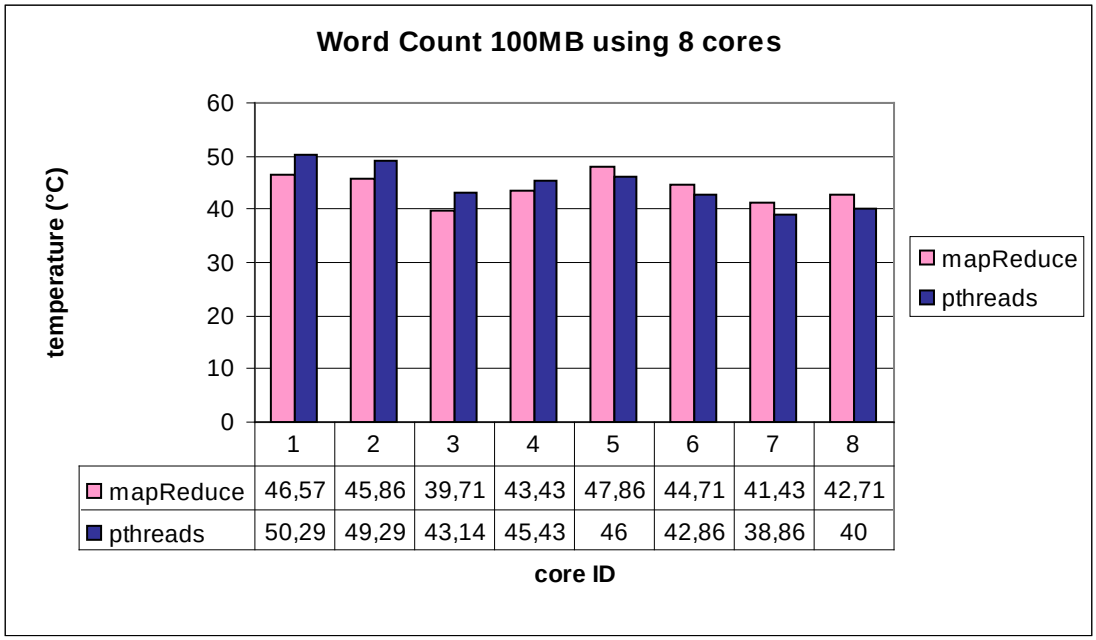
Word Count – Αρχείο μεγέθους 100MB

Στις πιο κάτω γραφικές (σχήμα 6.2.5.ιιι) παρουσιάζονται οι γραφικές για τα 4 σενάρια της εφαρμογής Word Count με αρχείο μεγέθους 100MB.





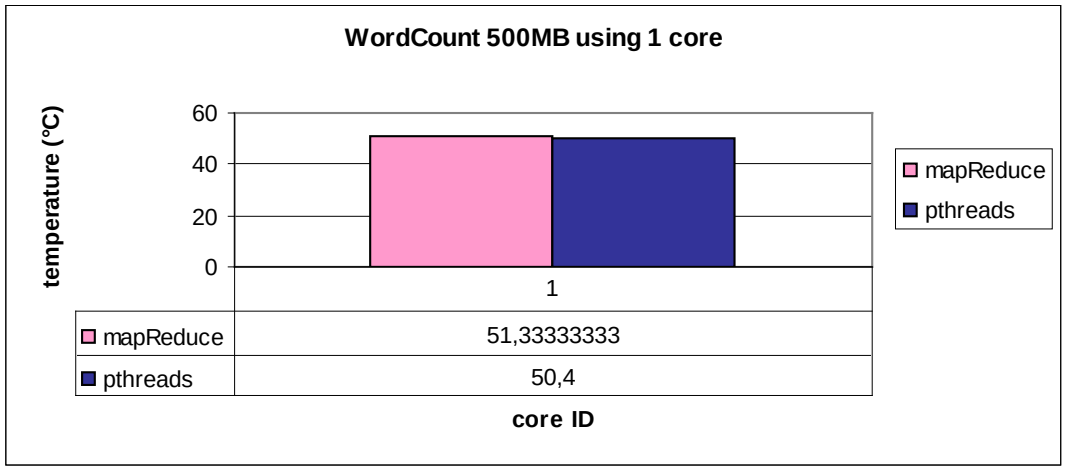


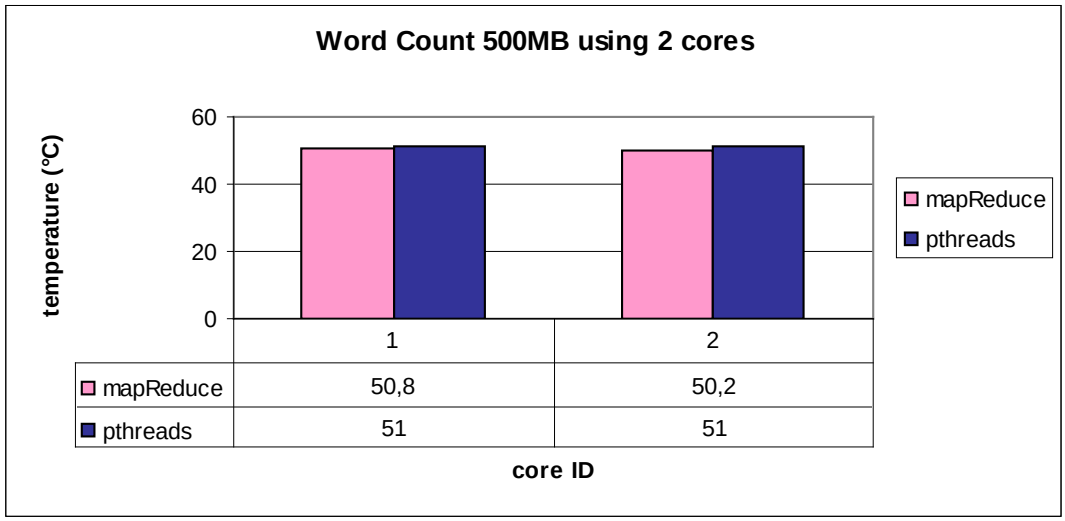


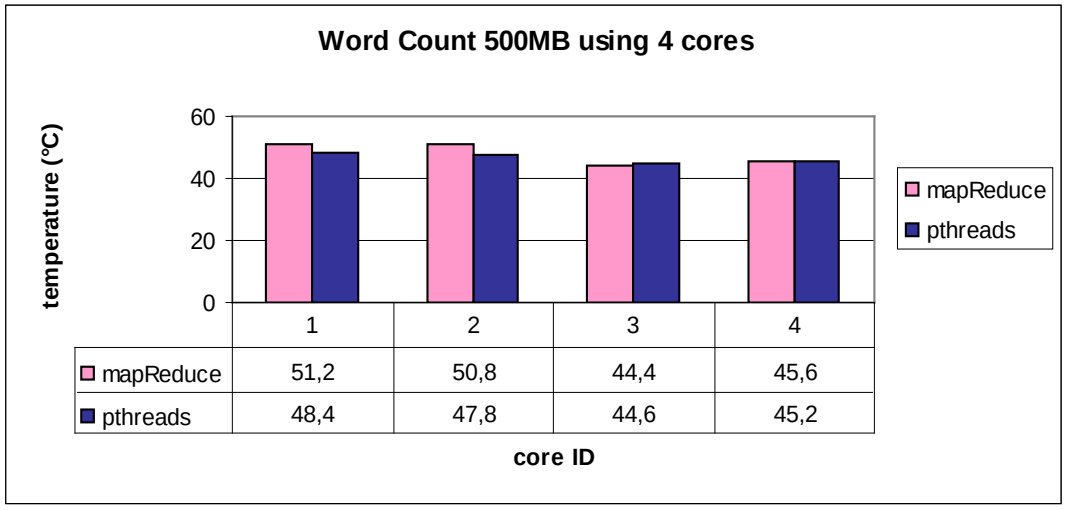
Σχήμα 6.2.5.ιιι: Γραφικές παραστάσεις εφαρμογής Word Count με αρχείο μεγέθους 100MB

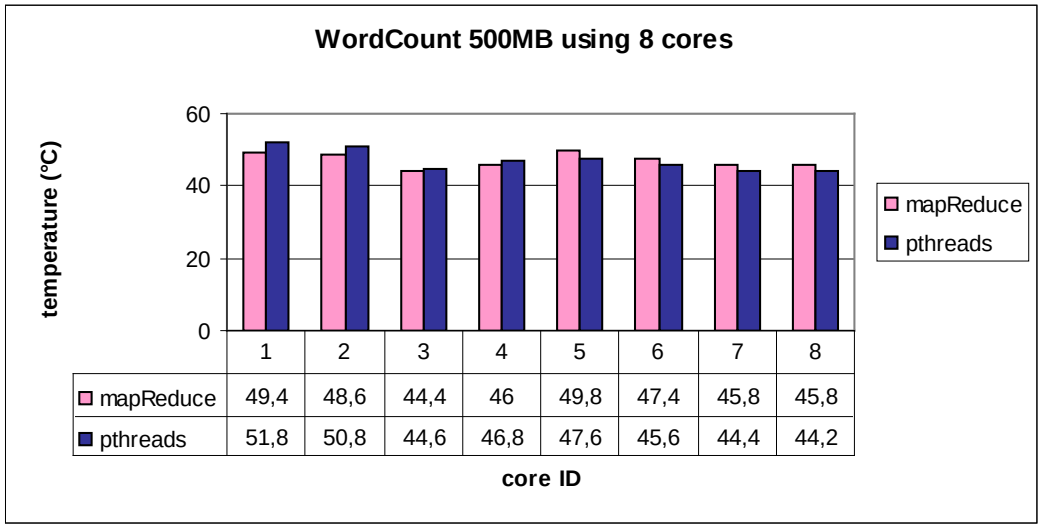
Word Count – Αρχείο μεγέθους 500MB

Στις πιο κάτω γραφικές (σχήμα 6.2.5.ιιιι) παρουσιάζονται οι γραφικές για τα 4 σενάρια της εφαρμογής Word Count με αρχείο μεγέθους 500MB.


WordCount 500MB using 1 core
temperature (°C)
60
40
20
0
1
mapReduce
51,33333333
pthreads
50,4
core ID
mapReduce
pthreads


Word Count 500MB using 2 cores
temperature (°C)
60
40
20
0
1
2
mapReduce
50,8
50,2
pthreads
51
51
core ID
mapReduce
pthreads

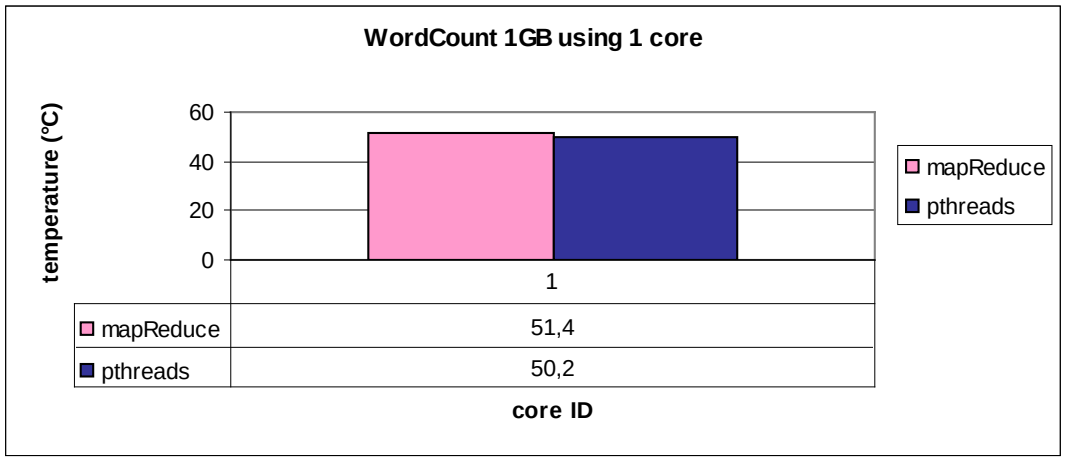

Word Count 500MB using 4 cores
temperature (°C)
60
40
20
0
1
2
3
4
mapReduce
51,2
50,8
44,4
45,6
pthreads
48,4
47,8
44,6
45,2
core ID
mapReduce
pthreads

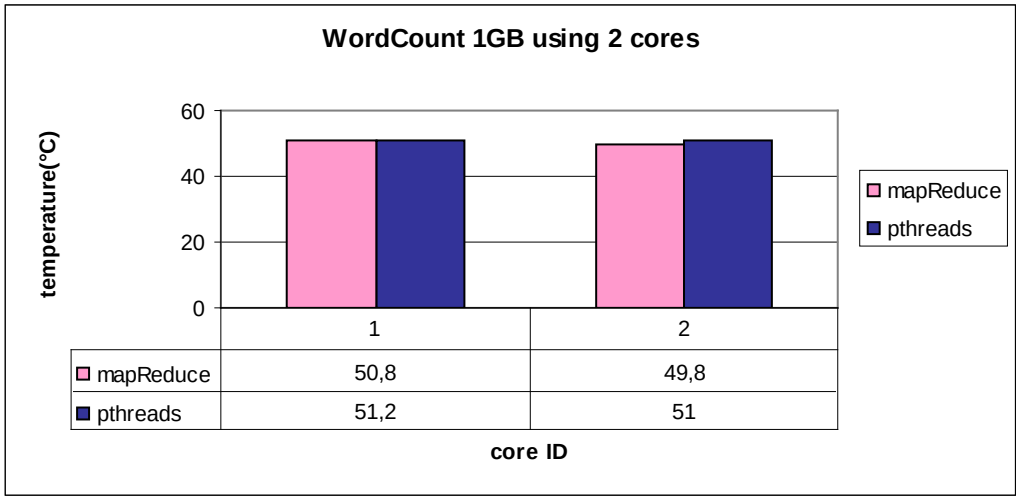


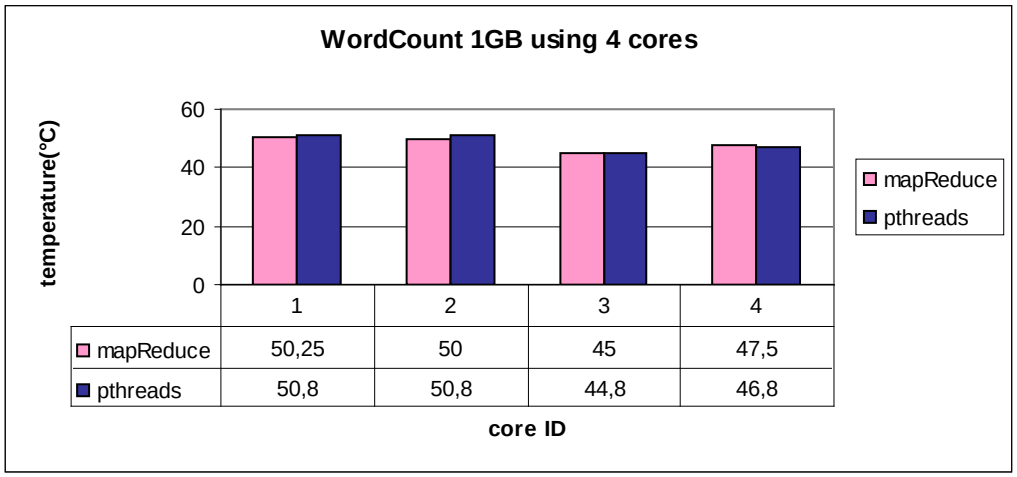
Σχήμα 6.2.5.ιιι: Γραφικές παραστάσεις εφαρμογής Word Count με αρχείο μεγέθους 500MB

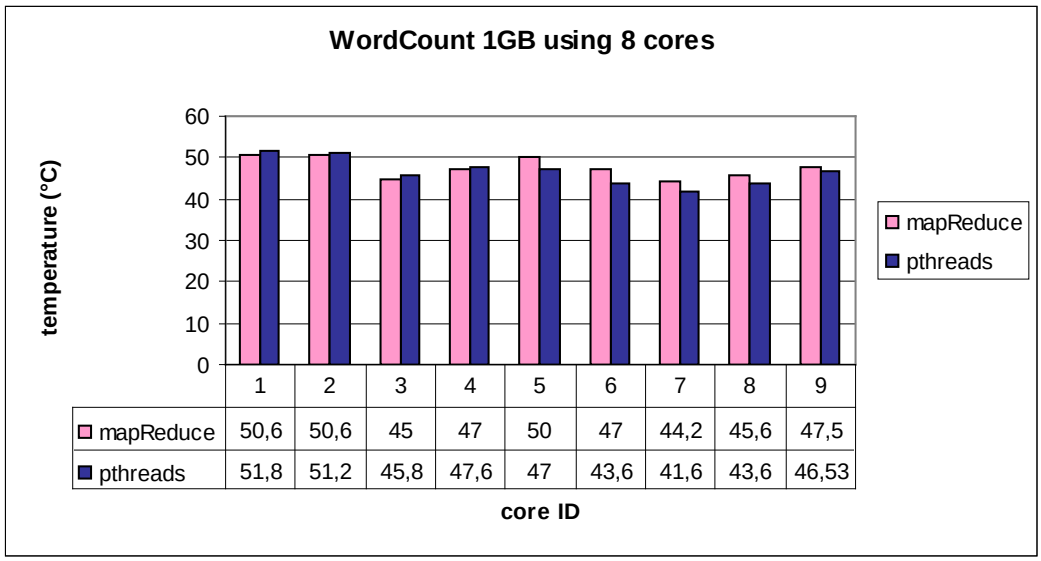
Word Count – Αρχείο μεγέθους 1GB

Στις πιο κάτω γραφικές (σχήμα 6.2.5.ιιιι) παρουσιάζονται οι γραφικές για τα 4 σενάρια της εφαρμογής Word Count με αρχείο μεγέθους 1GB.









Σχήμα 6.2.5.ιιιι: Γραφικές παραστάσεις εφαρμογής Word Count με αρχείο μεγέθους 1GB

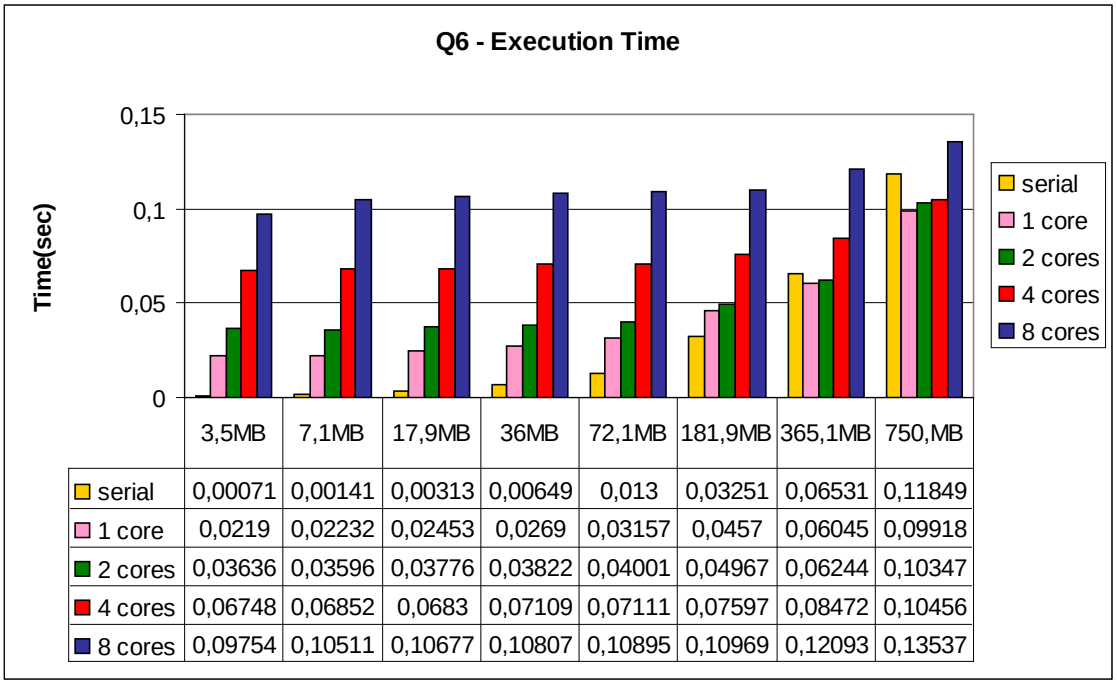
Στα σχήματα 6.2.5.ι, 6.2.5.ιι, 6.2.5.ιιι, 6.2.5.ιιιι και 6.2.5.ιιιιι παρουσιάζεται ο μέσος όρος των τιμών στην θερμοκρασία κάθε πυρήνα όπως έχουν καταγραφεί από την εκτέλεση της εφαρμογής Word Count. Τα αποτελέσματα είναι σε σύγκριση με τα αντίστοιχα αποτελέσματα της ίδιας εφαρμογής υλοποιημένης σε pthreads.

Όπως και οι δύο προηγούμενες εφαρμογές string match και linear regression, έτσι και η εφαρμογή word count έτρεξε για αρχεία διαφορετικού μεγέθους και στις δύο υλοποιήσεις. Και εδώ συγκρίνοντας σε κάθε περίπτωση τις δύο υλοποιήσεις phoenix και pthreads δεν παρουσιάζονται διαφορές ανάμεσα στις τιμές που καταγράφηκαν για τη θερμοκρασία. Συγκρίνοντας και πάλι την υλοποίηση του Phoenix όταν τρέχει με τον ίδιο αριθμό πυρήνων για διαφορετικό μέγεθος αρχείου εισόδου, παρατηρείται κάποια αύξηση στην θερμοκρασία αλλά ελάχιστα πιο μεγάλη. Κάτι το οποίο οφείλεται και πάλι στο περισσότερο χρόνο που χρειάζεται η εφαρμογή για να τρέξει για μεγαλύτερο όγκο αρχείου.

6.3 Εφαρμογές TCP-H

Για το query 6, εφαρμογή την οποία υλοποίησα από το TPC-H χρειάστηκε να τρέξω για 8 διαφορετικά αρχεία εισόδου με διαφορετικό μέγεθος 4 σενάρια, ουσιαστικά για 1, 2, 4 και 8 πυρήνες. Για κάθε περίπτωση έχω τρέξει την εφαρμογή 20 φορές καταγράφοντας το χρόνο εκτέλεσης του. Στη συνέχεια αφαίρεσα το μεγαλύτερο και το μικρότερο χρόνο εκτέλεσης και υπολόγισα το μέσο όρο όσων τιμών απέμειναν.

Πιο κάτω, στο σχήμα 6.3 παραθέτω τη γραφική παράσταση με το χρόνο εκτέλεσης για τα τέσσερα σενάρια για κάθε διαφορετικό αρχείο δεδομένων εισόδου. Τα πιο κάτω αποτελέσματα είναι σε σύγκριση με την αντίστοιχη υλοποίηση σε σειριακό κώδικα.



	3,5MB	7,1MB	17,9MB	36MB	72,1MB	181,9MB	365,1MB	750,MB
serial	0,00071	0,00141	0,00313	0,00649	0,013	0,03251	0,06531	0,11849
1 core	0,0219	0,02232	0,02453	0,0269	0,03157	0,0457	0,06045	0,09918
2 cores	0,03636	0,03596	0,03776	0,03822	0,04001	0,04967	0,06244	0,10347
4 cores	0,06748	0,06852	0,0683	0,07109	0,07111	0,07597	0,08472	0,10456
8 cores	0,09754	0,10511	0,10677	0,10807	0,10895	0,10969	0,12093	0,13537

Σχήμα 6.3: Γραφική Παράσταση χρόνου εκτέλεσης εφαρμογής TPC-H – Q6.

Όπως μπορούμε να παρατηρήσουμε για μικρά αρχεία εισόδου ο σειριακός κώδικας έχει πολύ μικρότερο χρόνο εκτέλεσης από την αντίστοιχη παράλληλη υλοποίηση σε MapReduce. Όταν ο όγκος του αρχείου με τα δεδομένα εισόδου αρχίζει να μεγαλώνει τότε βλέπουμε τη διαφορά στο χρόνο εκτέλεσης της σειριακής υλοποίησης και την υλοποίησης στο Phoenix να μειώνεται, ενώ όσο συνεχίζει να αυξάνεται ο όγκος του αρχείου αρχίζει η υλοποίηση στο Phoenix να τρέχει πιο γρήγορα από τη σειριακή. Αυτό οφείλεται στην καλή διαχείριση του runtime system του Phoenix σε μεγάλο όγκο δεδομένων από ότι σε μικρό όγκο δεδομένων.

6.4 Περίληψη Αποτελεσμάτων

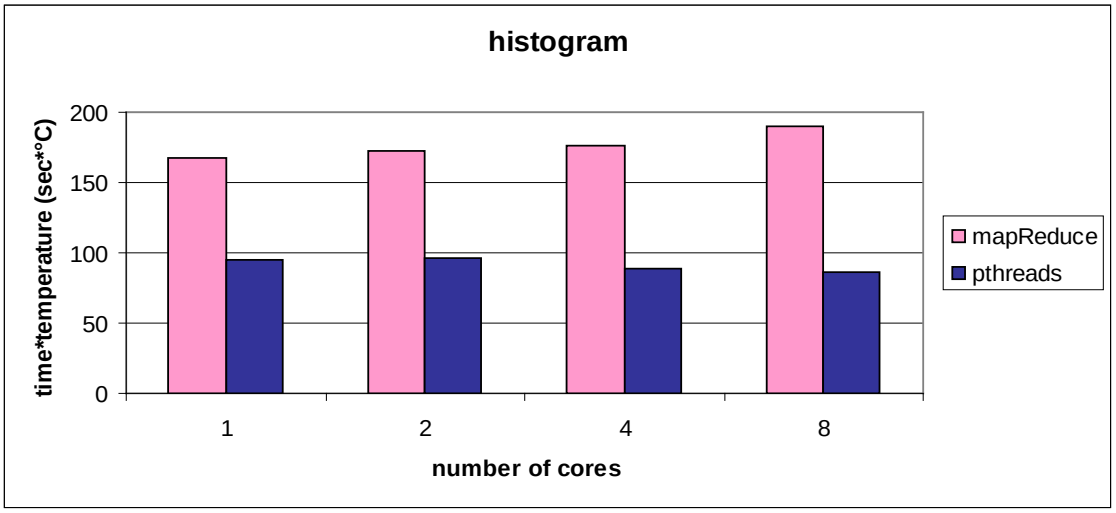
Τα αποτελέσματα τα οποία παραθέτω στο Κεφάλαιο 6.2 παρουσιάζουν τις μετρήσεις στη θερμοκρασία όπως καταγράφονται από τις εφαρμογές στο μοντέλο MapReduce για την υλοποίηση του Phoenix, σε σύγκριση με τις μετρήσεις που καταγράφονται από τις αντίστοιχες εφαρμογές υλοποιημένες σε pthreads.

Για να μπορέσουμε να εξάγουμε κάποια συμπεράσματα χρειάζεται να παραθέσω τα αποτελέσματα που προκύπτουν όταν υπολογίσω το time*temperature των εφαρμογών

που είναι υλοποιημένες στο προγραμματιστικό μοντέλο MapReduce και των αντίστοιχων εφαρμογών σε Pthreads. Με αυτό τον υπολογισμό (time*temperature) λαμβάνουμε υπόψη το χρόνο και την θερμοκρασία που λαμβάνουμε από κάθε σενάριο. Η πιο αποδοτική υλοποίηση σε κάθε περίπτωση είναι η υλοποίηση η οποία δίνει την μικρότερη τιμή κατά τον πιο πάνω υπολογισμό. Αυτό σημαίνει ότι χρειάστηκε να πάρω μετρήσεις και για το χρόνο εκτέλεσης κάθε εφαρμογής, πέρα από τις μετρήσεις των τιμών της θερμοκρασίας. Και πάλι χρειάστηκε να αφαιρέσω, από τις δέκα εκτελέσεις κάθε εφαρμογής, τον πιο μεγάλο χρόνο εκτέλεσης και τον πιο μικρό και να υπολογίσω το μέσο όρο όσων απομένουν. Επίσης ανάλογα με τον αριθμό των πυρήνων που χρειάζεται κάθε σενάριο για να τρέξει, όπως έχω περιγράψει προηγουμένως, υπολογίζω το μέσο όρο των τιμών της θερμοκρασίας.

Πριν προχωρήσω στις αντίστοιχες γραφικές αξίζει να παρατηρήσουμε ότι η θερμοκρασία που καταγράφτηκε για τη εφαρμογές οι οποίες υλοποιήθηκαν στο Phoenix σε σχέση με τις αντίστοιχες εφαρμογές υλοποιημένες σε Pthreads κυμαίνονται σε παρόμοια επίπεδα, χωρίς σημαντικές διαφορές μεταξύ τους. Επομένως, ουσιαστικά στη σύγκριση ανάμεσα στον πολλαπλασιασμό του χρόνου εκτέλεσης και της θερμοκρασίας στο Phoenix και τον πολλαπλασιασμό του χρόνου εκτέλεσης και της θερμοκρασίας σε Pthreads, θα παίξει σημαντικό ρόλο ο χρόνος εκτέλεσης κάθε υλοποίησης.

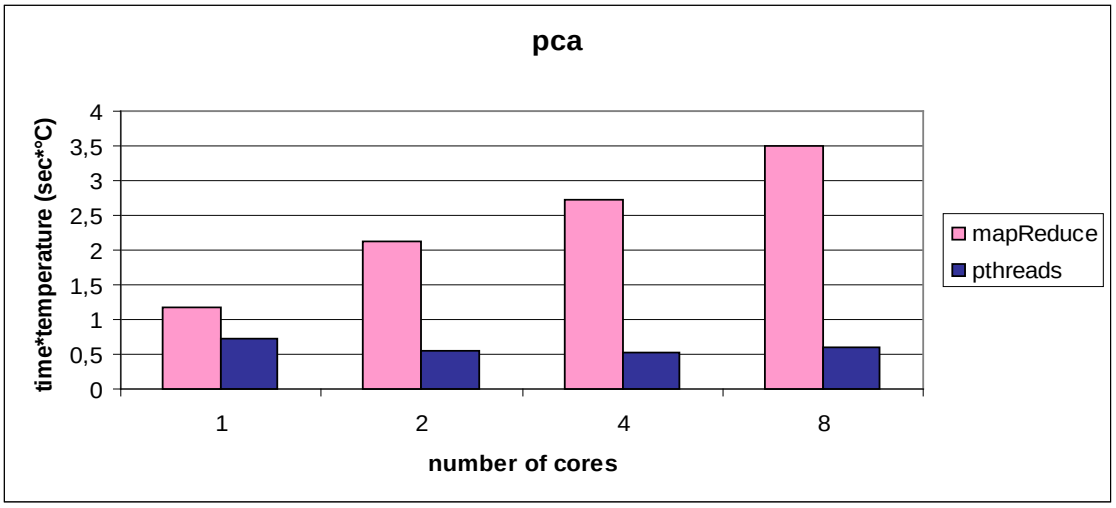
Histogram



Σχήμα 6.4.1: Γραφική Παράσταση Χρόνου Εκτέλεσης*Θερμοκρασίας της εφαρμογής Histogram

Όπως παρατηρούμε από το σχήμα 6.4.1, για τη συγκεκριμένη εφαρμογή, δίνει καλύτερα αποτελέσματα η υλοποίηση σε pthreads. Αυτό οφείλεται στη διαφορά του χρόνου εκτέλεσης των δύο υλοποιήσεων. Η υλοποίηση στο Phoenix χρειάζεται περισσότερο χρόνο εκτέλεσης ενώ η θερμοκρασία του κυμαίνεται σε πολύ παρόμοιες τιμές, με αποτέλεσμα η υλοποίηση σε pthreads να είναι πιο αποδοτική.

PCA

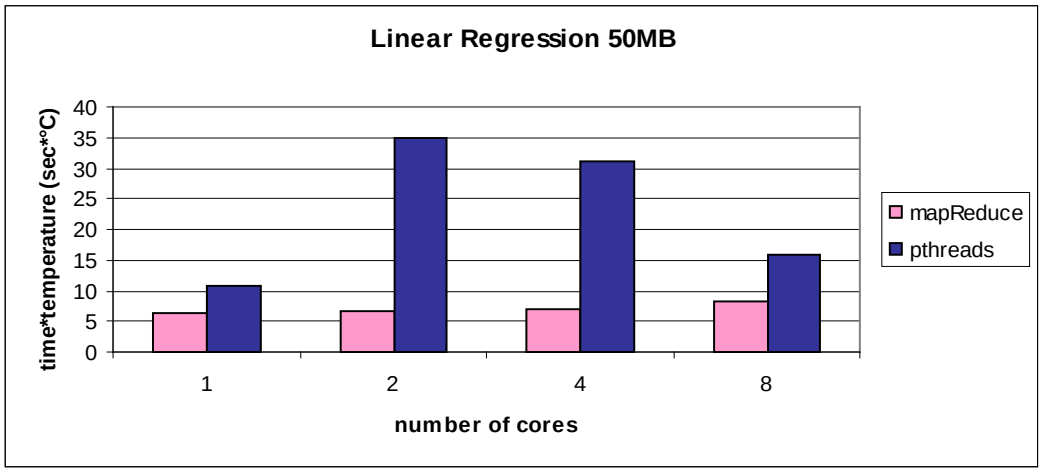


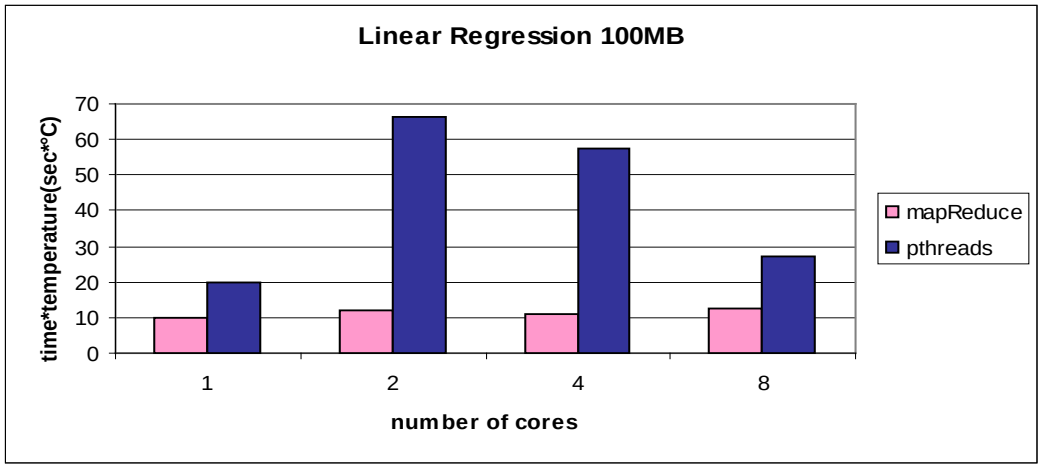
Σχήμα 6.4.2: Γραφική Παράσταση Χρόνου Εκτέλεσης*Θερμοκρασίας της εφαρμογής PCA

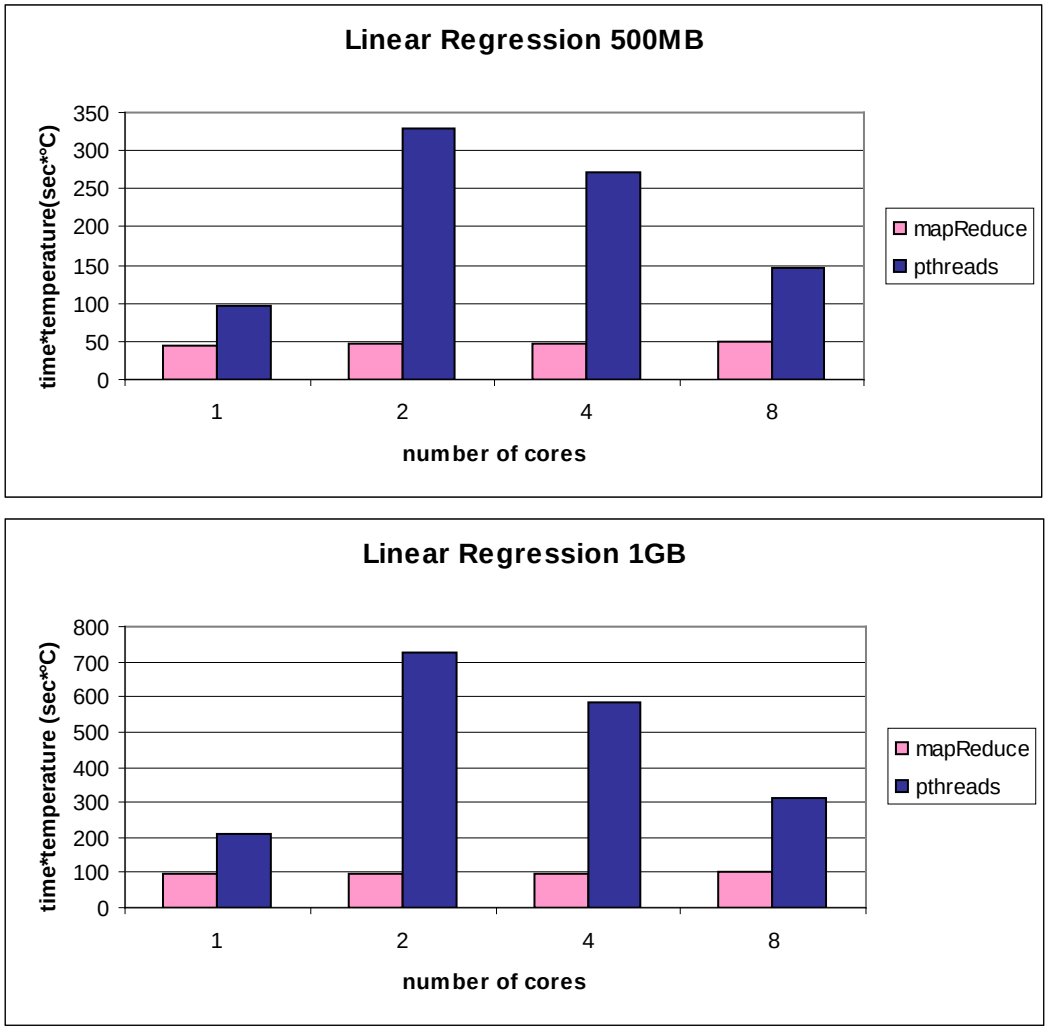
Και πάλι από το σχήμα 6.4.2, θα παρατηρήσουμε παρόμοια συμπεριφορά με την εφαρμογή histogram. Η εφαρμογή PCA δίνει καλύτερα αποτελέσματα με την υλοποίηση σε pthreads, αφού και πάλι η υλοποίηση στο Phoenix χρειάζεται περισσότερο χρόνο εκτέλεσης, ενώ η θερμοκρασία και στις δύο περιπτώσεις είναι παρόμοια κάνοντας την υλοποίηση σε pthreads πιο αποδοτική.

Οι δύο εφαρμογές που παρουσιάστηκαν πιο πάνω, η histogram και η PCA, στηρίζουν τη λειτουργία τους περισσότερο σε πολύπλοκους υπολογισμούς, παρά στην ανάγνωση μεγάλου όγκου δεδομένων εισόδου, κάτι το οποίο δεν μπορεί να διαχειριστεί αποδοτικά το runtime system του Phoenix αφού προκαλεί καθυστερήσεις μέχρι να αποφασίσει πως θα διαχωριστούν τα δεδομένα εισόδου στις map και στις reduce εργασίες.

Linear Regression



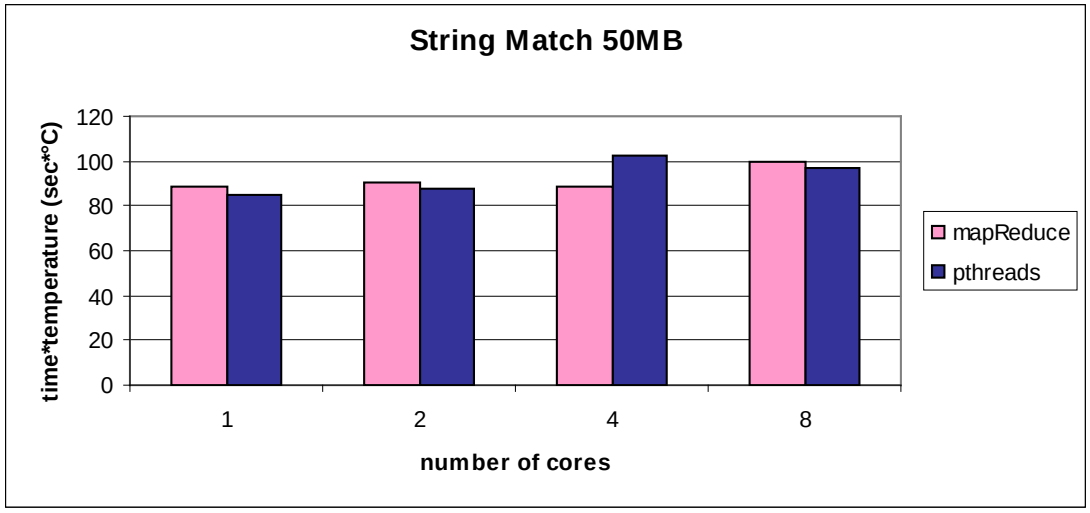


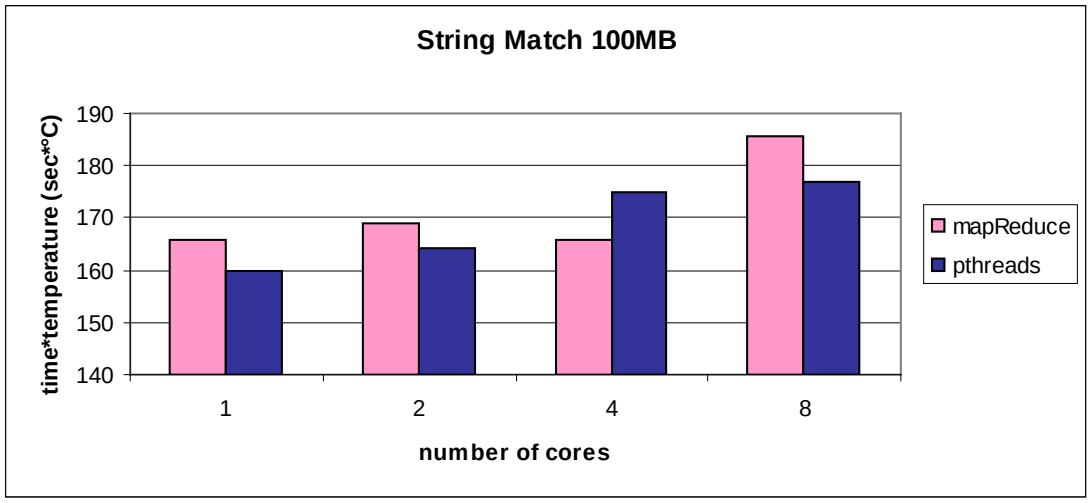


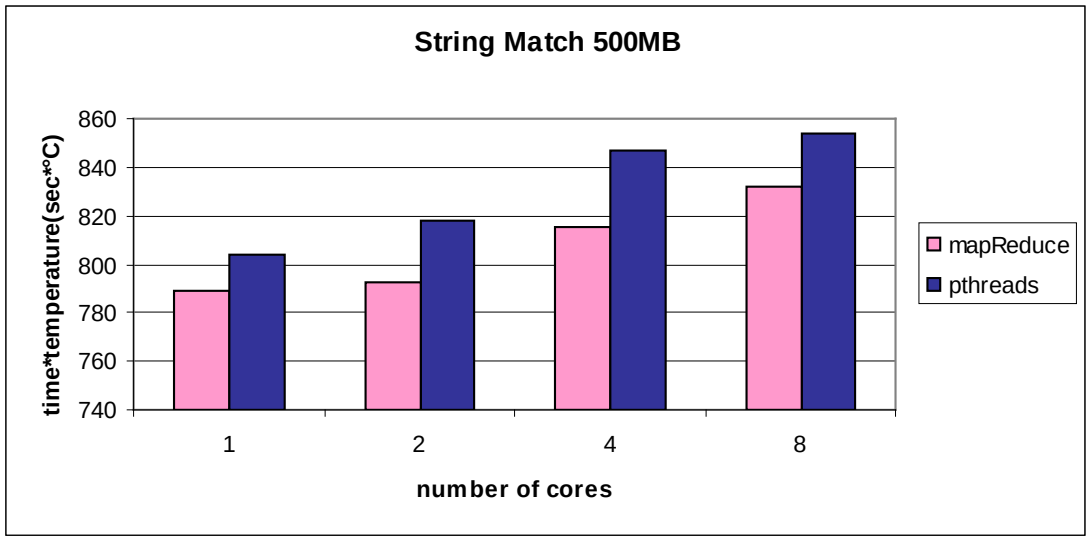
Σχήμα 6.4.3: Γραφικές Παραστάσεις Χρόνου Εκτέλεσης*Θερμοκρασίας της εφαρμογής Linear Regression για τα 4 διαφορετικά μεγέθη αρχείων

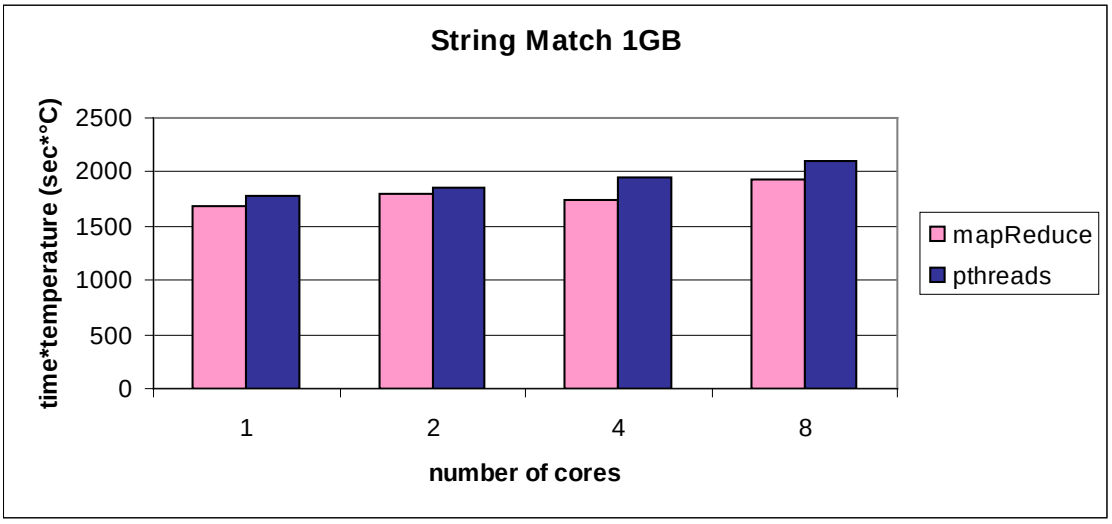
Παρατηρώντας με προσοχή τις 4 γραφικές παραστάσεις θα διαπιστώσουμε ότι όσο αυξάνεται το μέγεθος του αρχείου και ο όγκος των δεδομένων, δείχνει να είναι καλύτερη η υλοποίηση της εφαρμογής στο Phoenix. Παρατηρείται αύξηση ανάμεσα στη διαφορά του υπολογισμού time*temperature της υλοποίησης στο Phoenix και της υλοποίησης σε Pthreads, κάτι το οποίο θέτει προβάδισμα στο Phoenix για την συγκεκριμένη εφαρμογή, κυρίως σε συνδυασμό με μεγάλο όγκο δεδομένων. Αυτό οφείλεται στην αποδοτική διαχείριση αρχείων με μεγάλο όγκο δεδομένων από το runtime system, η οποία γίνεται δυναμικά σε αντίθεση με τη υλοποίηση σε pthreads στην οποία γίνεται στατικά.

String Match







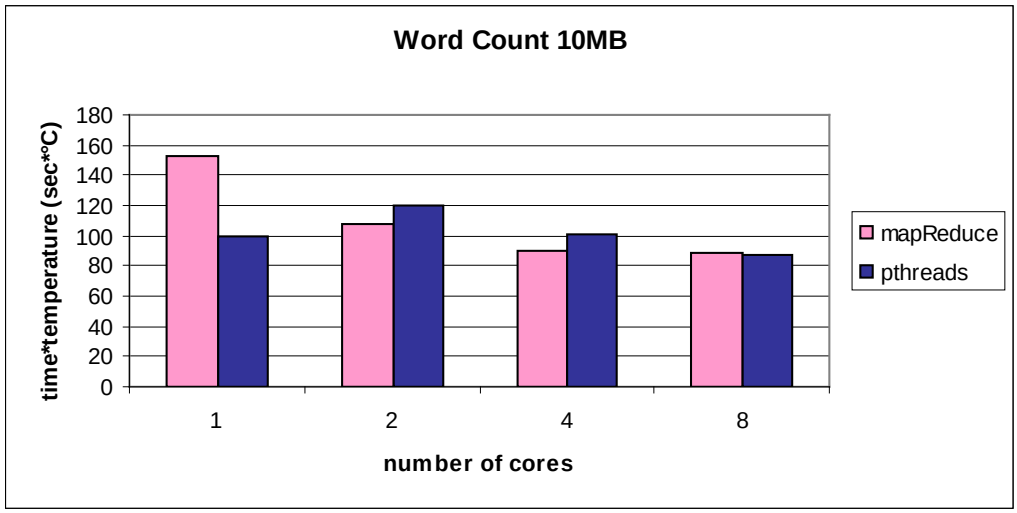


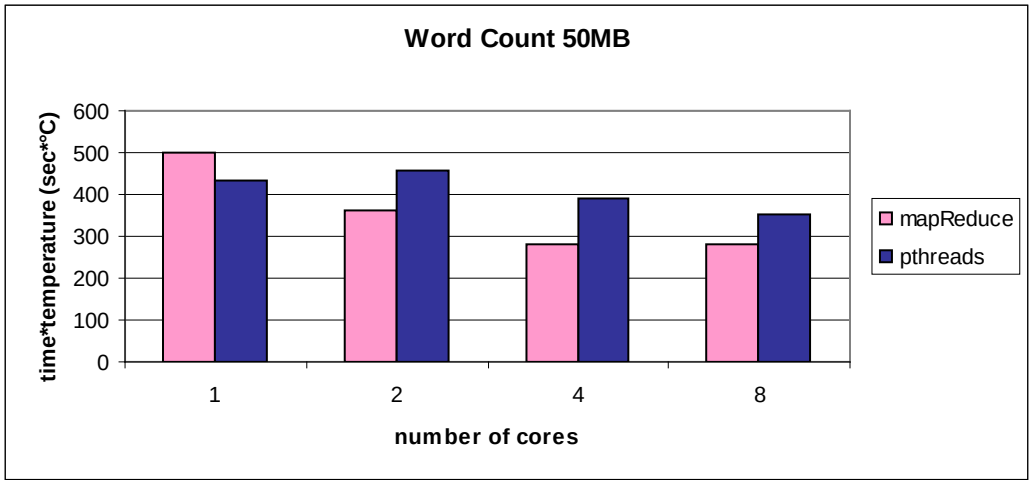
Σχήμα 6.4.4: Γραφικές Παραστάσεις Χρόνου Εκτέλεσης*Θερμοκρασίας της εφαρμογής String Match για τα 4 διαφορετικά μεγέθη αρχείων.

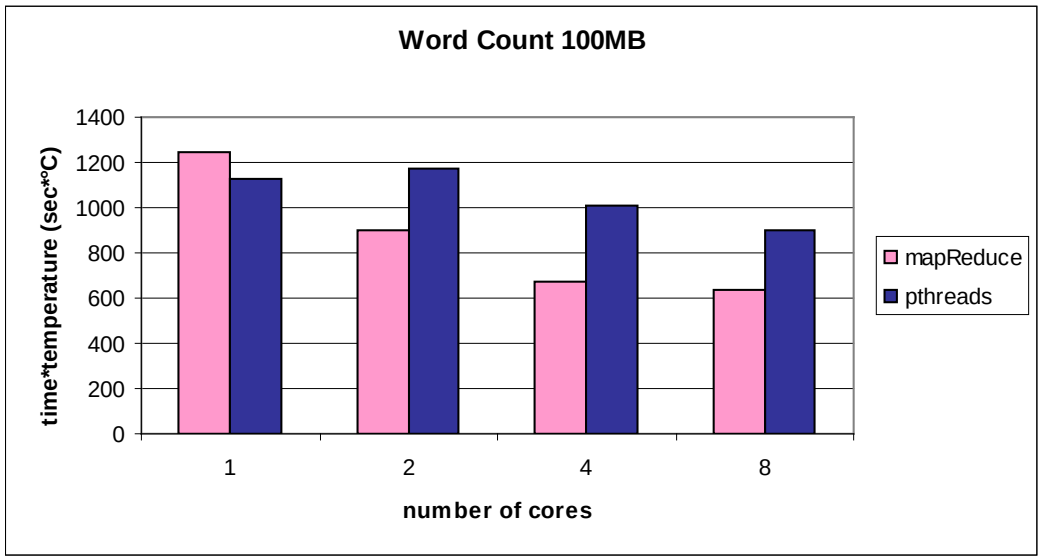
Σε αυτή την εφαρμογή παρατηρούμε τις περισσότερες φορές την υλοποίηση του Phoenix να έχει μικρότερο λόγο σε σχέση με την υλοποίηση σε Pthreads. Και πάλι, όπως συμβαίνει και στην εφαρμογή Linear Regression, παρατηρούμαι ότι όταν αυξάνεται το μέγεθος του αρχείου η υλοποίηση σε mapReduce έχει σταθερά το προβάδισμα, και στα 4 σενάρια με διαφορετικό αριθμό πυρήνων.

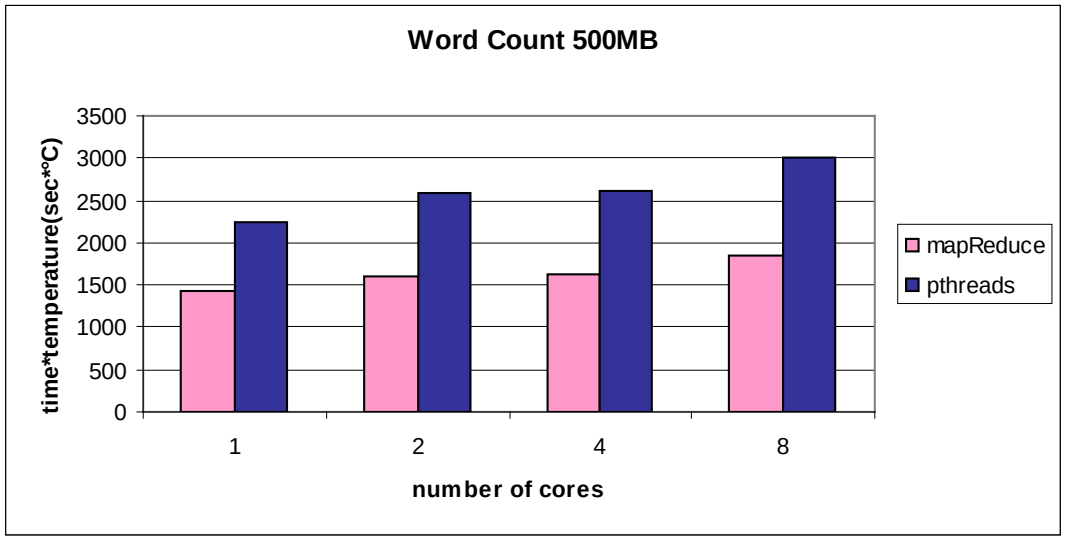
Ουσιαστικά και στις τέσσερις περιπτώσεις χρειάζεται λιγότερο χρόνο εκτέλεσης σε σχέση με τα Pthreads και αυτό οφείλεται και πάλι στη αποδοτική διαχείριση του runtime system του Phoenix για αρχεία με μεγάλο όγκο δεδομένων. Μέσα από τη συγκεκριμένη εφαρμογή φαίνεται το runtime system του Phoenix να έχει μια μικρή αδυναμία στη διαχείριση αρχείων με μικρό όγκο δεδομένων και στη χρονοδρομολόγηση των διαφόρων εργασιών σε σχέση με την υλοποίηση σε pthreads.

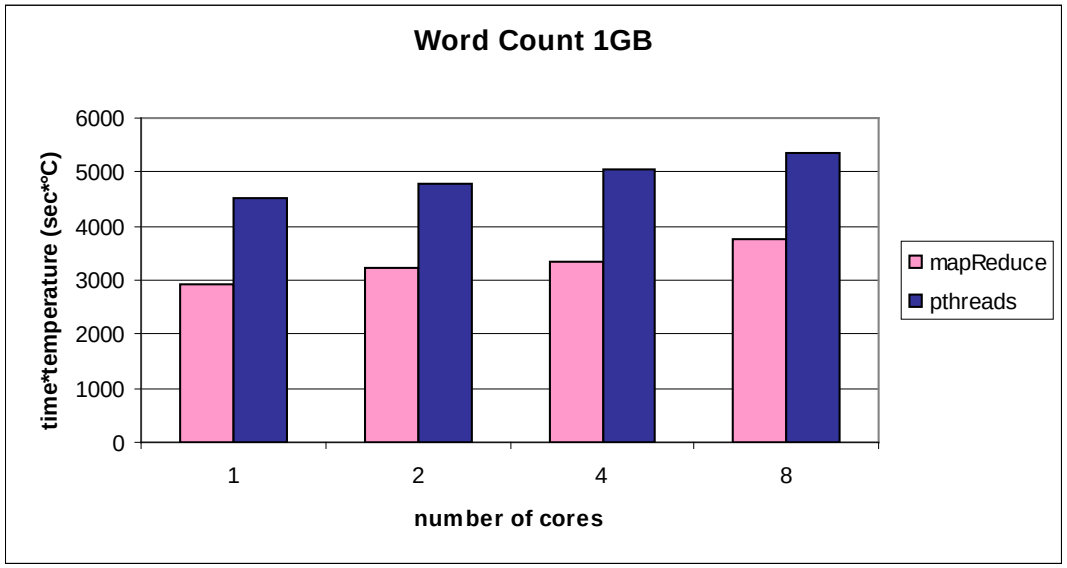
Word Count


Word Count 10MB
time*temperature (sec*°C)
180
160
140
120
100
80
60
40
20
0
1
2
4
8
number of cores
mapReduce
pthreads


Word Count 50MB
time*temperature (sec*°C)
600
500
400
300
200
100
0
1
2
4
8
number of cores
mapReduce
pthreads


Word Count 100MB
time*temperature (sec*°C)
1400
1200
1000
800
600
400
200
0
1
2
4
8
number of cores
mapReduce
pthreads





Σχήμα 6.4.5: Γραφικές Παραστάσεις Χρόνου Εκτέλεσης/Θερμοκρασίας της εφαρμογής Word Count για τα 5 διαφορετικά μεγέθη αρχείων.

Στη συγκεκριμένη εφαρμογή, παρατηρούμαι ότι στις πλείστες περιπτώσεις για τα 5 αρχεία δεδομένων εισόδου μικρότερη τιμή στον υπολογισμό time*temperature δίνει η υλοποίηση του Phoenix. Η διαφορά ανάμεσα στις δύο υλοποιήσεις είναι πιο εμφανής στις περιπτώσεις όπου αρχίζει να αυξάνεται ο όγκος του αρχείου με τα δεδομένα εισόδου. Αυτό δείχνει και πάλι ότι το runtime system του Phoenix, διαχειρίζεται δεδομένα εισόδου με μεγάλο όγκο πολύ αποδοτικά χωρίς να προκαλούνται οποιεσδήποτε καθυστερήσεις στην εκτέλεση της εφαρμογής.

Γενικά η υλοποίηση του Phoenix παρουσιάζεται λίγο καλύτερη στις εφαρμογές οι οποίες έχουν να κάνουν κυρίως με δεδομένα με μεγάλο όγκο δεδομένων. Ουσιαστικά υπάρχουν εφαρμογές οι οποίες μπορούν να προσαρμοστούν εύκολα στο μοντέλο MapReduce. Στο μοντέλο αυτό ταιριάζουν εφαρμογές στις οποίες τα δεδομένα συσχετίζονται πάντα με κάποιο κλειδί ή εφαρμογές στις οποίες το να γίνει εισαγωγή κάποιου κλειδιού σε ένα μεγάλο block δεδομένων δεν επιβαρύνει την εφαρμογή με επιπλέον καθυστερήσεις οι οποίες να οφείλονται σε επιπλέον υπολογισμούς μέσω των οποίων θα σχηματιστεί το ζεύγος(κλειδί/τιμή).

Το Phoenix λειτουργεί με pointers, χωρίς να χρειάζεται να αντιγράφει όλα τα δεδομένα που χρειάζεται κάθε φορά, κάτι το οποίο οδηγεί σε μείωση των καθυστερήσεων κυρίως κατά το στάδιο διαχωρισμού του αρχείου εισόδου σε κομμάτια. Επίσης, σημαντικό ρόλο στην επίδοση των εφαρμογών του Phoenix, παίζει το πως οι ίδιες οι εφαρμογές θα χειριστούν το δυναμικό scheduling που τους παρέχεται.

Στο σχήμα 6.4.1 και 6.4.2 παρατηρήσαμε ότι οι δύο εφαρμογές δεν είχαν καλά αποτελέσματα στην εφαρμογή με την υλοποίηση του Phoenix. Για τα συγκεκριμένα αποτελέσματα στην εφαρμογή του histogram ευθύνεται το γεγονός ότι η υλοποίηση σε Pthreads μειώνει τις καθυστερήσεις, αφού δεν χρησιμοποιεί κλειδιά όπως αναμένεται από το output format, ενώ η ταξινόμηση πάνω στα τελικά αποτελέσματα αποφεύγεται. Από την μεριά την υλοποίηση της εφαρμογής PCA στο Phoenix, αυξάνει τις καθυστερήσεις γιατί δεν χρησιμοποιεί το original array structure και έτσι πρέπει να κρατά τις συντεταγμένες για κάθε data point ξεχωριστά. Αυτό έχει σαν αποτέλεσμα για κάθε integer από το input set να πρέπει να χειρίζεται άλλους δύο integers. Σε αντίθεση, η αντίστοιχη υλοποίηση σε Pthreads αποφεύγει τις επιπλέον καθυστερήσεις, αφού έχει απευθείας πρόσβαση στον πίνακα δεδομένων.

Τα πιο πάνω αποτελέσματα αποδεικνύουν ότι το MapReduce μοντέλο είναι λίγο καλύτερο σε εφαρμογές οι οποίες περιέχουν υπολογισμούς οι οποίοι είναι απλό να εκφραστούν σε κώδικα, και ο χειρισμός τους γίνεται δυναμικά χωρίς να χρειάζεται κάποια ιδιαίτερη προσπάθεια από τον προγραμματιστή.

Κλείνοντας, να αναφερθώ στα αποτελέσματα της υλοποίησης του query 6 που έφτιαξα στο Phoenix. Από τη δική μου υλοποίηση, σε σύγκριση με το σειριακό κώδικα φαίνεται ότι στις περισσότερες περιπτώσεις ο σειριακός κώδικας τρέχει σε πολύ λιγότερο χρόνο. Τα αποτελέσματα πιθανόν να οφείλονται σε διάφορες καθυστερήσεις, κατά το διαχωρισμό του αρχείου με τα δεδομένα εισόδου, ή σε καθυστερήσεις που προκαλούνται κατά τη συνάρτηση map, όπου ορίζονται τα ζεύγη. Κοιτάζοντας με προσοχή το σχήμα 6.3, θα παρατηρήσουμε ότι όταν αρχίζει να μεγαλώνει ο όγκος του αρχείου των δεδομένων εισόδου, η δική μου παράλληλη υλοποίηση στο Phoenix παρουσιάζει λιγότερο χρόνο εκτέλεσης σε σχέση με τη σειριακή υλοποίηση. Επίσης ο κώδικας, ο οποίος έχω υλοποιήσει δεν έχει τύχει βελτιστοποιήσεων οι οποίες θα τον βοηθούσαν να αποφύγει κάποιες καθυστερήσεις.

Κεφάλαιο 7

Συμπεράσματα και Μελλοντική Εργασία

7.1 Συμπεράσματα

Μετά την πειραματική αξιολόγηση αυτής της διπλωματικής εργασίας είμαι σε θέση να πω ότι για μηχανές μέχρι 8 πυρήνες, δεν παρατηρούνται σημαντικές διαφορές στην θερμοκρασία των πυρήνων του συστήματος εάν χρησιμοποιηθεί το προγραμματιστικό μοντέλο MapReduce στην υλοποίηση Phoenix σε σχέση με αντίστοιχες υλοποιήσεις σε Pthreads.

Επομένως, κάποιος ο οποίος χρειάζεται να υλοποιήσει μια εφαρμογή σε συστήματα πολυεπεξεργαστών και έχει να επιλέξει ανάμεσα στην υλοποίηση του Phoenix και στα Pthreads δεν χρειάζεται να ανησυχεί για την θερμοκρασία του συστήματος του όταν η μηχανή του αποτελείται από αριθμό πυρήνων μέχρι 8. Θα πρέπει να τον απασχολήσει όμως το θέμα του χρόνου εκτέλεσης που θα χρειάζεται η εφαρμογή ανάλογα με το μοντέλο, όπως επίσης και το μοντέλο στο οποίο μπορεί να προσαρμοστεί πιο εύκολα η εφαρμογή του.

Κατά την άποψη μου, το η υλοποίηση του Phoenix μπορεί να προταθεί σε εφαρμογές οι οποίες έχουν να κάνουν με μεγάλο όγκο δεδομένων εισόδου. Επίσης μπορούν να προσαρμοστούν σε αυτό το μοντέλο εφαρμογές που δέχονται δεδομένα εισόδου σε μορφή κειμένου, γιατί συνήθως δεδομένα αυτής της μορφής δεν χρειάζονται πολύπλοκους υπολογισμούς για να σχηματιστεί το ζεύγος (κλειδί/τιμή) μειώνοντας

έτσι τις καθυστερήσεις. Εφαρμογές με παρόμοια χαρακτηριστικά, αποτελούν εφαρμογές οι οποίες έχουν να κάνουν με ταξινόμηση και αναζήτηση.

Επίσης, το μοντέλο μπορεί να προταθεί σε περιπτώσεις στις οποίες ο προγραμματιστής επιθυμεί να γράψει πιο απλή μορφή κώδικα, χωρίς να χρειάζεται να ανησυχεί για λεπτομέρειες παραλληλισμού, για πιθανές βελτιστοποιήσεις στον κώδικα, για την ανοχή σφαλμάτων και την κατανομή των δεδομένων (load balancing) για τις διάφορες εργασίες.

Θεωρώ ότι η υλοποίηση του Phoenix, αφού δεν δημιουργεί προβλήματα με τη θερμότητα που παράγει στα συστήματα πολυεπεξεργαστών σε συνεργασία με την καλή επίδοση σε ορισμένες κατηγορίες εφαρμογών, θα παίξει πολύ σημαντικό ρόλο στο μέλλον του παράλληλου υπολογισμού.

7.2 Μελλοντική Εργασία

Το προγραμματιστικό μοντέλο MapReduce έχει προταθεί σχετικά πρόσφατα, και πιο συγκεκριμένα η υλοποίηση στα συστήματα πολυεπεξεργαστών. Υπάρχει ακόμη αρκετή δουλειά και έρευνα γύρω από το συγκεκριμένο προγραμματιστικό μοντέλο.

Μια μελλοντική εργασία είναι να εφαρμοστεί η εργασία που έγινε για τη συγκεκριμένη διπλωματική σε ένα προσομοιωτή στον οποίο θα μπορούν να τρέξουν οι εφαρμογές της υλοποίησης του Phoenix και να καταγράφονται οι τιμές της θερμοκρασίας του συστήματος. Το ιδιαίτερο με αυτή την πρόταση είναι το γεγονός ότι ο simulator θα μπορεί να προσημειώνει της εφαρμογές, έτσι ώστε να μπορούν να τρέχουν σε συστήματα με πολύ μεγαλύτερο αριθμό πυρήνων στο ίδιο κύκλωμα (chip). Μια τέτοια εργασία θα οδηγήσει σε συμπεράσματα σχετικά με το πόσο scalable είναι η υλοποίηση του MapReduce μοντέλου σε συστήματα πολυεπεξεργαστών, σε σχέση με τη θερμότητα που θα παράγεται.

Μια άλλη πρόταση είναι να γίνει μια σχετική έρευνα οι οποία να χρησιμοποιεί το μοντέλο MapReduce σε συστάδες υπολογιστές (clusters), οι οποίοι όμως θα

αποτελούνται από πολυπύρηνους επεξεργαστές. Έτσι θα καταλήξουμε σε συμπεράσματα κατά πόσο είναι αποδοτική η συνεργασία cluster αρχιτεκτονικής και αρχιτεκτονικής πολυεπεξεργαστών.

Τέλος, αντίστοιχη εργασία με τη δική μου θα μπορούσε να γίνει και στις άλλες αρχιτεκτονικές για τις οποίες έγιναν υλοποιήσεις για το MapReduce μοντέλο. Συγκεκριμένα θα μπορούσε να γίνει παρόμοια δουλειά και να καταγραφεί η θερμότητα που παράγεται σε συστάδα υπολογιστών (clusters), στις κάρτες γραφικών (GPUs) και στον CELL.

Βιβλιογραφία

[1] Bingsheng He, et al, “Mars: A MapReduce Framework on Graphics Processors”, in 17th international conference on Parallel architectures and compilation techniques, 2008.

[2] Colby Ramanan, et al, “Evaluating MapReduce for Multi-core and Multiprocessor Systems”, in IEEE 13th International Symposium on High Performance Computer Architecture, 2007.

[3] Jacob Leverich and Christos Kozyrakis, “On the Energy (In) efficiency of Hadoop Clusters”, presented on Workshop on power aware computing and systems (HotPower '09), 2010.

[4] Jeffrey Dean and Sanjay Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, Google, Inc, 2004.

[5] Marc de Kruijf and Karthikeyan Sankaralingam, “MapReduce for the Cell B.E Architecture”, 2007.

[6] Yanpei Chen, Laura Keys and Randy H. Katz, (August 2009),”Towards Energy Efficient MapReduce”, EECS Department University of California, Berkeley, Technical Report [online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-109.pdf

[7] The Phoenix System for MapReduce Programming, Stanford University, [online]. Available: http://mapreduce.stanford.edu/

[8] Hadoop – MapReduce Implementation, July 2009, [online]. Available: http://hadoop.apache.org/mapreduce/

[9] Lm_sensors – Linux Hardware Monitoring, October 2009, [online]. Available: http://www.lm-sensors.org/wiki

[10] TPC-H Benchmarks, May 2010, [online]. Available: http://www.tpc.org/tpch/default.asp

[11] Intel Xeon Processor – Thermal Management, May 2010, [online]. Available: http://www.intel.com/support/processors/xeon/sb/CS-007761.htm

[12] Joseph Jaja, “An Introduction to Parallel Algorithms”, Addison-Wesley, 1992

Παράρτημα Α

Παρουσίαση υλοποιημένων αλγορίθμων

Α.1 Ψευδοκώδικας εφαρμογής Word Count σε MapReduce

```
/* intermediate output: key=word; value=1*/
Map(void *input) {
        for each word w in input
        EmitIntermediate(w, 1);
}

/*intermediate output: key=word; value=1*/
//output: key=word; value=occurrences
Reduce(String key, Iterator values) {
        int result = 0;
        for each v in values
        result += v;
        Emit(w , result);
}
```

Α.2 Query 6 σε ψευδοκώδικα SQL

```
select
        sum(l_extendedprice*l_discount) as revenue
from
        lineitem
where
        l_shipdate >= date '[DATE]'
        and l_shipdate < date '[DATE]' + interval '1' year
        and l_discount between [DISCOUNT] - 0.01 and [DISCOUNT] + 0.01
        and l_quantity < [QUANTITY];
```

A.3 Υλοποίηση Query 6 σε MapReduce

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>

#include <strings.h>
#include <stddef.h>
#include <unistd.h>
#include <assert.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <ctype.h>
#include <inttypes.h>

#include "map_reduce.h"
#include "stddefines.h"


struct timeval t0, t1;
double dt;

int result_count = 0;

typedef struct
{
      float ext_price; //extended price
      float discount;  //quantity
      float year;           //year
      float quantity;

}q6_data_t;

#define NINPUTS 7

char *paths[]= {"../queryData/inp_0.01/",
                 "../queryData/inp_0.01/",
                "../queryData/inp_0.02/",
                "../queryData/inp_0.05/",
                "../queryData/inp_0.1/",
                "../queryData/inp_0.2/",
                "../queryData/inp_0.5/",
                "../queryData/inp_1/"};

int line_sizes[]=       {14,
                        60175,
                        120515,
                        299814,
                        600572,
```

```
                        1199969,
                        2999671,
                        6001215};

int order_sizes[]=      {150,
                        15000,
                        30000,
                        75000,
                        150000,
                        300000,
                        750000,
                        1500000};

int customer_sizes[]=   {15,
                        1500,
                        3000,
                        7500,
                        15000,
                        30000,
                        75000,
                        150000};

     void compute_average(float times[]);
     float times[10];

     int LINEITEM;
     #define LINEITEMX 10000000
     #define DATE1 1994
     #define DATE2 1995
     #define DISCOUNT 0.01
     #define QUANTITY 24
     #define ITERATIONS 20
     #define WARMUP_ITERATIONS 3
     #define LINE_MAX 200

     // DATE1 and DATE2 means years in float format, eg. 2007.0
     // DISCOUNT AND QUANTITY are other constants given by the
user


float line_item[LINEITEMX][4];

void q6_map(map_args_t *args)
{
  int cv01 = 0, count = 0;

  int temp_key[args->length];
  float temp_val[args->length];

  int* key;
  float* val;

  float sum = 0;
```

```
  assert(args);
  q6_data_t* data = (q6_data_t*)(args->data);
  assert(data);


  key = &(temp_key[0]);
  val = &(temp_val[0]);

while (cv01 < args->length)
{
     if((data[cv01].year>=DATE1) && (data[cv01].year<DATE2) &&
     (data[cv01].discount > (DISCOUNT -0.01)) &&
     ((float)(data[cv01].discount) < (float)(DISCOUNT+0.01)) &&
     (data[cv01].quantity < QUANTITY) )
     {
       sum = data[cv01].ext_price*data[cv01].year;
       count++;
       temp_key[cv01] = cv01 % 10 + 1;
       temp_val[cv01] = sum;
     emit_intermediate((void*)&key[cv01],(void*)&val[cv01],(siz
eof(int)));
     }
     ++cv01;
 }

}

void q6_reduce(void *key_in, iterator_t *itr)
{

    float *sumptr = CALLOC(sizeof(float), 1);
    register float sum = 0;
    float *val;
    short *key = (short *)key_in;
    int count =0;
    assert(key);
    assert(itr);

    while (iter_next (itr, (void **)&val))
    {
      sum += *val;
      count ++;
    }

    *sumptr = sum;
    emit(key, (void *)sumptr);
}

void *q6_combiner (iterator_t *itr)
{
    float *sumptr = CALLOC(sizeof(float), 1);
    register float sum = 0;
    float *val;
```

```
    assert(itr);
    while (iter_next (itr, (void **)&val))
    {
        sum += *val;
    }

    *sumptr = sum;
    return (void *)sumptr;
}

int mykeycmp(const void *s1, const void *s2)
{
    short val1 = *((short *)s1);
    short val2 = *((short *)s2);

    if (val1 < val2) {
        return -1;
    }
    else if (val1 > val2) {
        return 1;
    }
    else {
        return 0;
    }
}
int main(int argc, char* argv[])
{

  final_data_t q6_vals;
  double process_data_time=0.0;
  int num_processor;

  char buff[256];
  float sum=0;
  int k=0;
  int i=0;//,j=0;
  int count = 1;
  FILE *fp;
  char line[LINE_MAX];
  char *year, *month, *day;
  char *orderkey, *quantity, *extendedprice, *discount,
*shipdate, *commitdate, *receiptdate, *shipmode;


  if (argc != 4)
  {
        printf("./program input fileName processors_num\n");
        return 0;
  }
  char* coreIdFname;
  coreIdFname = argv[2];
  num_processor=atoi(argv[3]);

  FILE* tempFile;
```

```
  tempFile = fopen(coreIdFname,"a");
fprintf(tempFile,"**********************************************
*********\n");
  fclose(tempFile);

  result_count = 0;

  LINEITEM = line_sizes[atoi( argv[1] )];
  printf( "LINEITEM = %d\n", LINEITEM );

  strcpy(buff, paths[atoi( argv[1] )]);
  strcat(buff, "lineitem.tbl");
  fp=fopen(buff,"r"); /*Open the file Input*/
  if ( fp == NULL ) {
      fprintf( stderr, "Error opening lineitem.tbl\n" );
      exit(-1);
  }

while (( fgets ( line, sizeof line, fp) != NULL ) &&
(count<LINEITEM)){
      orderkey = strtok(line, "|");
      quantity = strtok(NULL, "|");
      extendedprice = strtok(NULL, "|");
      discount = strtok(NULL, "|");
      shipdate = strtok(NULL, "|");
      commitdate = strtok(NULL, "|");
      receiptdate = strtok(NULL, "|");
      shipmode = strtok(NULL, "|");

      year = strtok(shipdate, "-");
      month = strtok(NULL, "-");
      day = strtok(NULL, "-");

      line_item[count][0]=atof(extendedprice); //extended price
      line_item[count][1]=atof(discount);  //discount
      line_item[count][2]=atof(year);       //year
      line_item[count][3]=atof(quantity);  //quantity


      count+=1;
}
  fclose(fp);

  LINEITEM=count;

  // In the future this data will be loaded from the reference
data set of TPC-H
  for(k=0;k<WARMUP_ITERATIONS;++k){
      for(i=0;i<LINEITEM;++i){

            if((line_item[i][2]>=DATE1) &&
      (line_item[i][2]<DATE2) && ((line_item[i][1]>(DISCOUNT-
```

```
      0.01)) && (line_item[i][1]<(DISCOUNT+0.01))) &&
      (line_item[i][3]<QUANTITY)){

                  sum+=line_item[i][0]*line_item[i][2];

            }
      }
}

  q6_data_t* q6_data;

  q6_data = (q6_data_t*)malloc((count) * sizeof(q6_data_t));

  for(k=0; k<count; k++)
  {
     q6_data[k].ext_price = line_item[k][0];
     q6_data[k].discount = line_item[k][1];
     q6_data[k].year = line_item[k][2];
     q6_data[k].quantity = line_item[k][3];
  }

  for(k=0;k<ITERATIONS;++k){

     //Time - Start
     gettimeofday(&t0, NULL);


    CHECK_ERROR (map_reduce_init ());

    // Setup map reduce args
    map_reduce_args_t map_reduce_args;
    memset(&map_reduce_args, 0, sizeof(map_reduce_args_t));
    map_reduce_args.task_data = q6_data;
    //map_reduce_args.task_data = &(line_item);
    map_reduce_args.map = q6_map;
    map_reduce_args.reduce = q6_reduce;
    map_reduce_args.combiner = q6_combiner;
    map_reduce_args.splitter = NULL;
    map_reduce_args.key_cmp = mykeycmp;
    map_reduce_args.unit_size = sizeof(float);
    map_reduce_args.partition = NULL;
    map_reduce_args.result = &q6_vals;
    map_reduce_args.data_size = sizeof(float)*LINEITEM;
    map_reduce_args.L1_cache_size =
atoi(GETENV("MR_L1CACHESIZE"));

    map_reduce_args.num_map_threads =
atoi(GETENV("MR_NUMTHREADS"));

    map_reduce_args.num_reduce_threads =
atoi(GETENV("MR_NUMTHREADS"));

    map_reduce_args.num_merge_threads =
atoi(GETENV("MR_NUMTHREADS"));
```

```
    map_reduce_args.num_procs = num_processor;

    if (num_processor > sysconf(_SC_NPROCESSORS_ONLN))
    {
      printf("Error at line : \n");
      printf("\targs->num_procs > num_procs\n");
      exit (1);
    }

    map_reduce_args.key_match_factor =
(float)atof(GETENV("MR_KEYMATCHFACTOR"));

    CHECK_ERROR( map_reduce (&map_reduce_args, coreIdFname) <
0);

    CHECK_ERROR (map_reduce_finalize ());

    sum=0.0;

    for (i = 0; i < q6_vals.length; i++)
    {
          keyval_t * curr = &((keyval_t *)q6_vals.data)[i];
         sum+=(*(float*)curr->val);
    }

   // Time - End
  gettimeofday(&t1, NULL);
  dt = t1.tv_sec - t0.tv_sec;
  if (t1.tv_usec >= t0.tv_usec)
    dt = dt + (t1.tv_usec - t0.tv_usec) * 1e-6;
  else
    dt = (dt-1) + (1e6 + t1.tv_usec - t0.tv_usec) * 1e-6;

  process_data_time+=dt;

}
    printf("sum = %f \n", sum);
    printf("Time: %f seconds.\n",process_data_time/ITERATIONS);

  return 0;
}
```