Ατομική Διπλωματική Εργασία

A TFlux implementation of MapReduce software

ΣΚΟΥΡΟΣ ΝΙΚΟΣ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΜΑΙΟΣ 2012

Department of Computer Science - Τμήμα Πληροφορικής
University of Cyprus - Πανεπιστήμιο Κύπρου

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

A TFlux implementation of MapReduce software

Σκούρος Νίκος

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 2012

Ευχαριστίες

Ολοκληρώνοντας τις προπτυχιακές μου σπουδές στο τμήμα Πληροφορικής του Πανεπιστημίου Κύπρου, θα ήθελα να ευχαριστήσω όλους εκείνους που συνέβαλαν στην υλοποίηση της ατομικής διπλωματικής μου εργασίας.

Πρώτα από όλα θα ήθελα να ευχαριστήσω τους επιβλέποντες καθηγητές μου κ.κ. *Pedro Trancoso* (Professor Ph.D., University of Illinois at Urbana-Champaign, USA, 1998) καθώς και τον *Διαβαστό Ανδρέα* (PhD student) που έχει προηγούμενη πείρα και συμμετοχή στο project “TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems” και *Κωνσταντίνο Χριστοφή* (PhD student) που είχε αναπτύξει παλιότερα διπλωματική εργασία με θέμα “Evaluating Hadoop, a MapReduce implementation,for Multicore systems” που μου έδωσαν την ώθηση και τις γνώσεις για να ολοκληρώσω την διπλωματική μου εργασία. Στην συνέχεια θα ήθελα να ευχαριστήσω όλους τους καθηγητές μου που στάθηκαν αρωγοί στις προσπάθειές μου και με εμπλούτισαν με τις πολύτιμες γνώσεις τους.

Έπειτα θα ήθελα να ευχαριστήσω την οικογένειά μου για την αμέριστη συμπαράσταση (τόσο οικονομική όσο και ηθική) για όλη τη χρονική περίοδο των σπουδών μου.

Τέλος θα ήθελα να ευχαριστήσω όλους τους φίλους μου με τους οποίους συνεργάστηκα όλα αυτά τα χρόνια, όχι μόνο τους συμφοιτητές μου, αλλά και ανθρώπους έξω από την σχολή. Δεν θα αναφερθώ ονομαστικά, με τον φόβο της παράλειψης κάποιου.

Κύπρος, 15η Μαΐου 2012

Περιεχόμενα

Περίληψη

Δεδομένου ότι η κατανάλωση ρεύματος (και κατά συνέπεια και η παραγωγή θερμότητας) από τους υπολογιστές έχει προκαλέσει ανησυχία τα τελευταία χρόνια , η παράλληλη επεξεργασία έχει γίνει το κυρίαρχο πρότυπο στην αρχιτεκτονική υπολογιστών, κυρίως με τη μορφή των πολύ-πύρηνων επεξεργαστών.

Οι παράλληλοι υπολογιστές μπορούν να κατατάσσονται ανάλογα με το επίπεδο στο οποίο το υλικό (hardware) υποστηρίζει παραλληλισμό, με *multi-core* και πολλαπλών επεξεργαστών ηλεκτρονικούς υπολογιστές που έχουν πολλά στοιχεία επεξεργασίας μέσα σε ένα μοναδικό μηχάνημα, ενώ οι *clusters,* MPPs, και τα δίκτυα χρησιμοποιούν πολλαπλούς υπολογιστές να εργάζονται στον ίδιο έργο. Οι εξειδικευμένες παράλληλες αρχιτεκτονικές υπολογιστών μερικές φορές χρησιμοποιούνται παράλληλα με τους παραδοσιακούς επεξεργαστές, για την επιτάχυνση συγκεκριμένων καθηκόντων.

Γνωστά και ευρέως χρησιμοποιημένα είναι ήδη ορισμένα μοντέλα επεξεργασίας δεδομένων όπως τα pipelines και οι ουρές μηνυμάτων. Αυτά τα μοντέλα παρέχουν συγκεκριμένες δυνατότητες στην ανάπτυξη εφαρμογών επεξεργασίας δεδομένων . Αξιέπαινα και ευέλικτα μοντέλα είναι αυτά τα οποία πρόβαλε η Google και το Πανεπιστήμιο Κύπρου , το MapReduce και TFlux αντίστοιχα.

Το MapReduce είναι ένα προγραμματιστικό μοντέλο και μια εφαρμογή για την επεξεργασία και την παραγωγή μεγάλου όγκου δεδομένων. Τα τελευταία χρόνια παρατηρείται η ανάγκη διαχείρισης όλο και μεγαλύτερου όγκου δεδομένων πληροφοριών(άνθρωποι ανεβάζουν βίντεο στο διαδίκτυο, βγάζουν φωτογραφίες μέσω των κινητών τους τηλεφώνων, αναβαθμίζουν τη σελίδα τους στο Facebook, αφήνουν σχόλια στο διαδίκτυο και τόσα πολλά) [17] [18] γεγονός που ανάγκασε μεγάλες εταιρίες του χώρου όπως οι Google, Yahoo!, Amazon και Microsoft να αναζητήσουν νέα εργαλεία για την επεξεργασία τέτοιων μεγάλων συνόλων δεδομένων. Η Google πρώτη δημοσίευσε το MapReduce ένα παράλληλο προγραμματιστικό μοντέλο για την κλιμάκωση των δεδομένων και αμέσως προσέλκυσε το ενδιαφέρον αρκετών εταιριών που αντιμετώπιζαν παρόμοια προβλήματα.

Η πλατφόρμα TFlux έχει δημιουργηθεί ούτος ώστε να επιτρέπει στους χρήστες της να εφαρμόζουν παράλληλο προγραμματισμό χρησιμοποιώντας Data-Driven Multithreading (DDM) μοντέλο εκτέλεσης το οποίο μπορεί να τρέξει αποδοτικά πάνω σε ένα σύστημα με πολλούς επεξεργαστές (multicores)

Στόχος

Στο πλαίσιο αυτής της Ατομική Διπλωματικής Εργασίας θα γίνει η προσπάθεια παραλληλισμού MapReduce υπολογισμών με την χρήση της πλατφόρμας TFlux και την αντιστοίχιση των δυο βασικών συναρτήσεων του μοντέλου, mapping και reducing, με ανάλογες συναρτήσεις που θα μπορούν να εκτελούνται παράλληλα . Ειδικότερα θα υλοποιηθούν κάποιες εφαρμογές σε γλωσσά προγραμματισμού C ,με κύριο συστατικό του παραλλήλου κώδικα τις TFlux Directives, και οι οποίες θα είναι αντίστοιχες με αυτές που χρησιμοποιεί η Google μέσο του μοντέλου MapReduce. Με το τρόπο αυτό θα υποδειχτεί πως είναι εφικτό να γίνει παραλληλισμός Map Reduce υπολογισμών με την χρήση της πλατφόρμας TFlux έτσι που να μπορεί να τρέξει αποδοτικά πάνω σε ένα σύστημα με πολλούς επεξεργαστές (multicore) και όχι πάνω σε μεγάλα clusters από εμπορικές μηχανές .

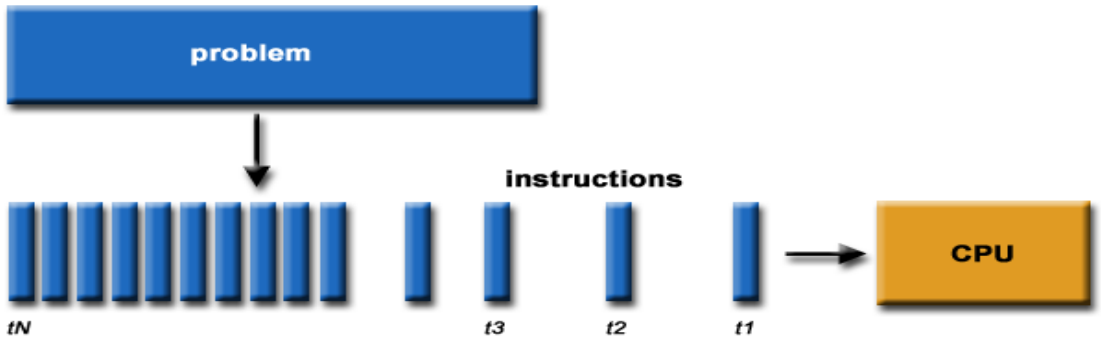
Για την επίτευξη του στόχου της διπλωματικής εργασίας είναι απαραίτητο να μελετηθεί το Phoenix, η υλοποίηση που έγινε για το προγραμματιστικό μοντέλο MapReduce πάνω σε συστήματα κοινόχρηστης μνήμης. Βάση αυτού του μοντέλου θα διερευνηθεί η λογική και μεθοδολογία παραλληλισμού MapReduce εφαρμογών σε πολυεπεξεργαστικά συστήματα . Επιπρόσθετα θα αποτελεί και το μοντέλο με το οποίο θα μπορούμε να συγκρίνουμε σε θέματα απόδοσης τις διαφορετικές υλοποιήσεις εφαρμογών με TFlux και MapReduce .

Επιπρόσθετα θα κατατεθεί η άποψη για το ποιο από τα προαναφερθέντα μοντέλα είναι πιο πρακτικό σε επίπεδο παραλληλισμού αλλά και τα μειονεκτήματα και πλεονεκτήματα τους . Τέλος θα εισηγηθούν πιθανές χρήσεις αυτής της δυνατότητας παραλληλισμού MapReduce εφαρμογών με TFlux και πως τα δυο μοντέλα αυτά μπορούν να συνδυαστούν με σκοπό την επίτευξη μεγίστου παραλληλισμού.

Κεφάλαιο 1

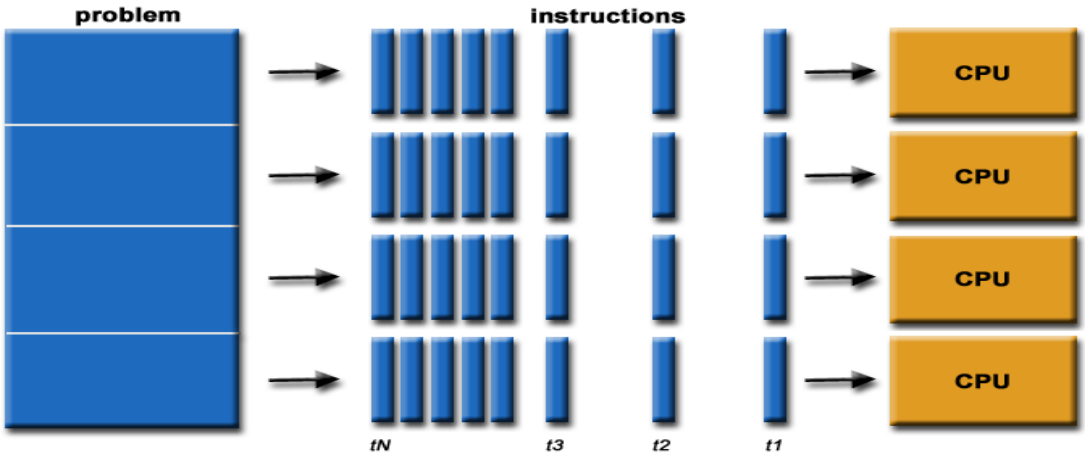
Εισαγωγή στην έννοια της παράλληλης επεξεργασίας

Τα παραδοσιακά συστήματα επεξεργασίας δεδομένων χρησιμοποιούν σειριακό υπολογισμό(serial computation),έτσι ώστε οι υπολογισμοί να εκτελούνται σε ένα μοναδικό υπολογιστή από μια Κεντρική Μονάδα Επεξεργασίας (ΚΜΕ), όπου το πρόβλημα διασπάται σε διακριτές ακολουθίες εντολών οι οποίες εκτελούνται σειριακά η μια μετά την άλλη , επιτρέποντας έτσι σε μια μόνο εντολή να εκτελείται κάθε χρονική στιγμή (t1, t2, t3, ..., tN).



Σχήμα 1.1 Σειριακή επεξεργασία [16]

Εντούτοις , τα σύγχρονα συστήματα υποκλίνονται σε μια νέα τάση , γνωστή ως «Παράλληλος Υπολογισμός». Ως παράλληλος υπολογισμός (parallel computing) ορίζεται η ταυτόχρονη χρήση πολλαπλών μονάδων επεξεργασίας με στόχο την αποδοτική επίλυση ενός υπολογιστικού προβλήματος . Στην απλούστερη μορφή της, η παράλληλη επεξεργασία είναι η ταυτόχρονη χρήση πολλαπλών υπολογιστικών πόρων για την επίλυση ενός προβλήματος .Οι υπολογισμοί εκτελούνται με την χρήση πολλαπλών Κεντρικών Μονάδων Επεξεργασίας , όπου το πρόβλημα διασπάται σε διακριτά τμήματα τα οποία μπορούν να εκτελούνται παράλληλα στις διαφορετικές ΚΜΕ. Για τις εντολές κάθε τμήματος ισχύουν οι περιορισμοί της ακολουθιακής εκτέλεσης(σειριακής).



Σχήμα 1.2 Παράλληλη εκτέλεση [16]

1.1 Γιατί Παράλληλη Επεξεργασία

Οι κύριοι λόγοι χρήσης της παράλληλης επεξεργασίας είναι η σημαντική μείωση του χρόνου εκτέλεσης των προγραμμάτων που τρέχουμε καθώς και η δυνατότητα επίλυσης μεγαλυτέρων προβλημάτων σε όγκο. Αυτό γίνεται στην πράξη πολύ αποδοτικότερα αφού παρέχεται συνδρομικότητα (ταυτόχρονης επεξεργασία). Μερικοί άλλοι λόγοι οι οποίοι είναι εξίσου σημαντικοί είναι η χρήση απομακρυσμένων πόρων μέσω τοπικών δικτύων και Διαδικτύου που αποφέρει και τη μείωση κόστους, αφού γίνεται χρήση πολλών "φθηνών" πόρων αντί λίγων και ακριβών (πχ υπέρ-υπολογιστών).Σημαντικό επίτευγμα θεωρείται επίσης το γεγονός ότι η κατανομή δεδομένων σε πολλούς υπολογιστές ώστε να ξεπεραστούν περιορισμοί στο μέγεθος της μνήμης.

1.2 Αρχιτεκτονική υλικού-λογισμικού παράλληλης επεξεργασίας

Οι υπολογιστικοί πόροι που χρησιμοποιούνται για την παράλληλη επεξεργασία μπορούν να περιλαμβάνουν είτε ένα μοναδικό υπολογιστή με πολλαπλούς επεξεργαστές *(multi-core computer) είτε* ένα αυθαίρετο αριθμό υπολογιστών συνδεδεμένων σε ένα κοινό δίκτυο (πολλαπλούς υπολογιστές κατανεμημένης μνήμης) η ακόμη και ένα συνδυασμό των δυο.

Το λογισμικό στην περίπτωση του παράλληλου προγραμματισμού συνήθως καταδεικνύει χαρακτηριστικά όπως η ικανότητα να είναι διασπασμένο σε διακριτά τμήματα εργασιών και τα οποία μπορούν να επιλυθούν ταυτόχρονα, να εκτελεί πολλαπλές εντολές προγράμματος ανά πάσα χρονική στιγμή και να μπορεί να επιλυθεί σε λιγότερο χρόνο χρησιμοποιώντας πολλαπλούς υπολογιστικούς πόρους παρά αντί ένα μοναδικό υπολογιστικό πόρο.

1.3 Υπολογιστές Μοιραζόμενης Μνήμης ή Πολύ-Επεξεργαστές

Τα παράλληλα συστήματα διακρίνονται σε πολυεπεξεργαστές κοινής μνήμης, όπου πολλαπλοί επεξεργαστές επικοινωνούν με μία κοινή μνήμη ενιαίου χώρου διευθύνσεων, και σε πολλαπλούς υπολογιστές κατανεμημένης μνήμης, όπου πολλαπλά πακέτα επεξεργαστή ιδιωτικής μνήμης, με τον δικό τους χώρο διευθύνσεων το καθένα, διασυνδέονται και επικοινωνούν μεταξύ τους. Και στις δύο περιπτώσεις η επικοινωνία γίνεται μέσω ενός «δικτύου διασύνδεσης» . Παρόλο που υπάρχουν διαφορές μεταξύ διαφορετικών αρχιτεκτονικών, γενικά όλοι οι επεξεργαστές έχουν πρόσβαση σε ένα καθολικό χώρο φυσικών διευθύνσεων. Οι επεξεργαστές εργάζονται ανεξάρτητα αλλά μοιράζονται τον ίδιο πόρο μνήμης . Τυχόν τροποποιήσεις σε περιοχές μνήμης που προκαλεί ένας επεξεργαστής είναι ορατές σε όλους τους επεξεργαστές.

1.4 Υπολογιστές Κατανεμημένης Μνήμης ή Πολυ-Υπολογιστές

Παρόμοια, οι αρχιτεκτονικές κατανεμημένης μνήμης διαφέρουν μεταξύ τους, αλλά παρουσιάζουν ένα κοινό χαρακτηριστικό όπου κάθε επεξεργαστής έχει τοπική (ιδιωτική) μνήμη ενώ η επικοινωνία με απομακρυσμένη μνήμη (τοπική μνήμη των άλλων επεξεργαστών) επιτυγχάνεται μέσω δικτύου και συνήθως με τη ρητή συμμετοχή του απομακρυσμένου επεξεργαστή. Ο κάθε επεξεργαστής έχει τον ιδιωτικό του χώρο διευθύνσεων, έτσι δεν υπάρχει η έννοια του καθολικού χώρου διευθύνσεων, κοινού για όλους τους επεξεργαστές. Αφού ο κάθε επεξεργαστής έχει τη δική του τοπική μνήμη, πιθανές τροποποιήσεις σε θέσεις μνήμης ενός επεξεργαστή δεν επηρεάζουν τη μνήμη άλλων επεξεργαστών. Η κρυφή μνήμη λειτουργεί ακριβώς όπως και στα συμβατικά συστήματα και δεν τίθεται ζήτημα συνοχής της κρυφής μνήμης. Όταν ένας επεξεργαστής πρέπει να προσπελάσει δεδομένα απομακρυσμένης μνήμης, συνήθως ο προγραμματιστής πρέπει ρητά να ορίσει πότε και ποιά δεδομένα θα προσπελαστούν. Παρόμοια ο συγχρονισμός μεταξύ παράλληλων εργασιών σε διαφορετικούς επεξεργαστές γίνεται ρητά από το πρόγραμμα.

1.5 Μοντέλα Παράλληλου Προγραμματισμού

Υπάρχουν αρκετά μοντέλα παράλληλου προγραμματισμού, τα επικρατέστερα όμως είναι δύο και αυτά είναι το Μοντέλο Νημάτων και το Μοντέλο Μεταβίβασης Μηνυμάτων. Εφαρμογή των δυο αυτών μοντέλων θα δούμε στη πλατφόρμα TFlux και MapReduce αντίστοιχα .Τα μοντέλα παράλληλου προγραμματισμού παρέχουν ένα επίπεδο αφαίρεσης μεταξύ των παράλληλων αλγορίθμων και των παράλληλων αρχιτεκτονικών, έτσι ώστε να είναι δυνατή η σχεδίαση και συγγραφή όσο το δυνατό πιο ευέλικτου και μεταφερτού κώδικα.

Κεφάλαιο 2^{ο} Η πλατφόρμα TFlux

2.1 Ορισμός πλατφόρμας TFlux

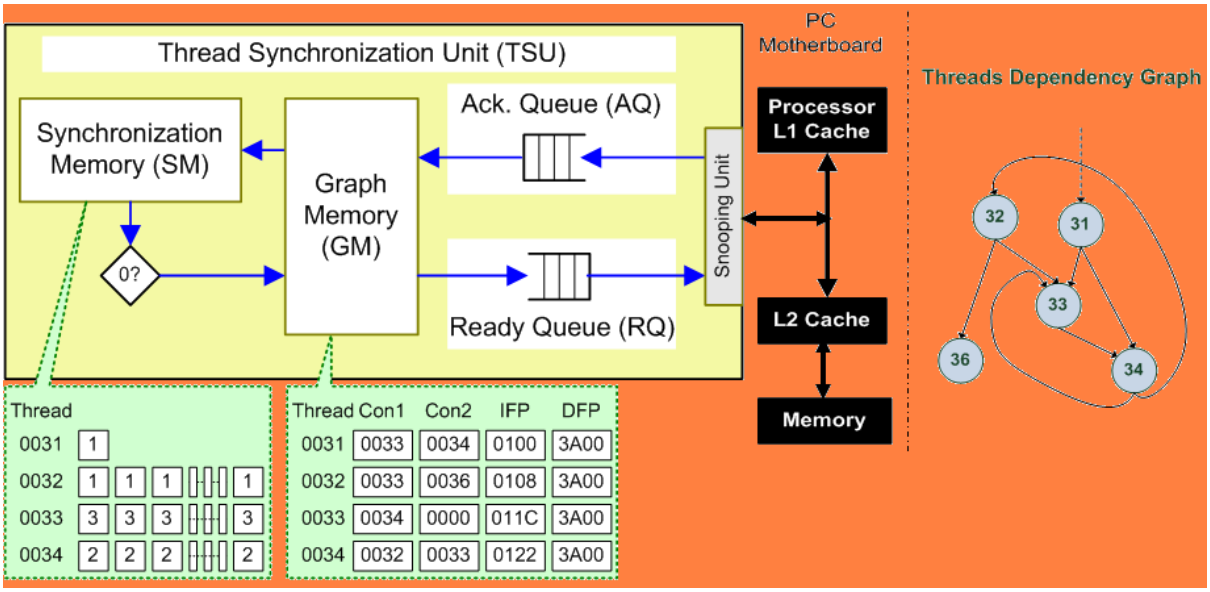
Η πλατφόρμα TFlux έχει δημιουργηθεί ούτος ώστε να επιτρέπει στους χρήστες της να εφαρμόζουν παράλληλο προγραμματισμό χρησιμοποιώντας Data-Driven Multithreading (DDM) μοντέλο εκτέλεσης το οποίο μπορεί να τρέξει αποδοτικά πάνω σε ένα σύστημα με πολλούς επεξεργαστές (multicores). Το TFlux δεν αναφέρεται σε κάποια υλοποίηση σε συγκεκριμένη μηχανή αλλά σε μια συλλογή από αφηρημένες οντότητες, οι οποίες επιτρέπουν την εκτέλεση προγραμμάτων χρησιμοποιώντας το μοντέλο DDM πάνω σε διάφορα υπολογιστικά συστήματα πολυεπεξεργασίας, ανεξάρτητα από τις λεπτομέρειες του υλικού ή τον τύπο του λειτουργικού συστήματος . Δηλαδή το TFlux παρέχει το επίπεδο virtualization για την εκτέλεση των TFlux προγραμμάτων κάτω από διαφορετικά συστήματα πολυεπεξεργασίας κοινής μνήμης

2.2 Παραλληλισμός σε επίπεδο νημάτων

Πολλές φορές σε έναν υπέρ-βαθμωτό επεξεργαστή ο οποίος εκτελεί ένα μόνο νήμα, είναι δυνατόν να μην μπορεί να γίνει χρήση όλων των διαθέσιμων πόρων τους συστήματος, γιατί δε θα είναι εφικτό να υπάρξει αρκετή παραλληλία σε επίπεδο εντολών (π.χ. όταν κάποια εντολή χρειάζεται να περιμένει δεδομένα από την κύρια μνήμη, όταν έγινε κάποια λανθασμένη πρόβλεψη διακλάδωσης, κ.α.). Αυτό οδηγεί σε σπατάλη υπολογιστικής δύναμης και δεν επιτρέπει την πλήρη εκμετάλλευση του υπολογιστικού συστήματος. Με την τεχνική SMT (Simultaneous MultiThreading– Ταυτόχρονος Πολυνηματισμός) ο επεξεργαστής μπορεί να εκτελεί την ίδια στιγμή εντολές από περισσότερα του ενός διαφορετικά νήματα εκτέλεσης . Αυτά τα νήματα θα ανταγωνίζονται μεταξύ τους για τους πόρους του επεξεργαστή, οι οποίοι θα μοιράζονται στα εν λόγω νήματα με στόχο να είναι συνεχώς κατειλημμένες όλες οι λειτουργικές μονάδες του. Με άλλα λόγια, αν ένα από τα νήματα που είναι προς εκτέλεση στον επεξεργαστή δεν μπορεί να εμφανίσει αρκετό παραλληλισμό σε επίπεδο εντολών ώστε να κρατά απασχολημένα συνεχώς όλα τα κυκλώματα του επεξεργαστή, παράλληλα με αυτό το νήμα θα μπορεί να εκτελείται και κάποιο άλλο, το οποίο θα μοιράζεται με το πρώτο τους πόρους του επεξεργαστή. Σε περίπτωση που υπάρχει ένα μοναδικό νήμα προς εκτέλεση, εάν δε μπορεί να απασχολεί συνέχεια τον επεξεργαστή, θα είναι αναπόφευκτη η σπατάλη πόρων, αλλά σε τέτοια περίπτωση δεν θα καθυστερεί κάποια άλλη εργασία.

2.3 Εκτέλεσης DDM Προγράμματος

Τα DDM προγράμματα αποτελούνται από δυο μέρη το κώδικα για τα DDM νήματα και το γράφο συγχρονισμού με τις εξαρτήσεις καταναλωτή-παραγωγού μεταξύ τους. Κατά την διαδικασία εκτέλεσης τους ο γράφος εξάρτησης φορτώνεται στην TSU για τον προγραμματισμό των νημάτων με βάση τα στοιχεία διαθεσιμότητας. Όταν ένα νήμα εκτελεστεί η TSU χρησιμοποιεί το γράφο για τον εντοπισμό και την ενημέρωση των καταναλωτών της. Μόλις παράγονται τα δεδομένα ενός νήματος καταναλωτή , η ενότητα CacheFlow στην TSU φορτώνει τα δεδομένα από την κύρια μνήμη στην cache του επεξεργαστή, όπου το νήμα καταναλωτής έχει προγραμματιστεί να εκτελεστεί. Μετά την φόρτωση του το νήμα είναι έτοιμο να ξεκινήσει.



Σχήμα 2.1 Εκτέλεσης DDM Προγράμματος [5]

2.4 Λειτουργία του TFlux

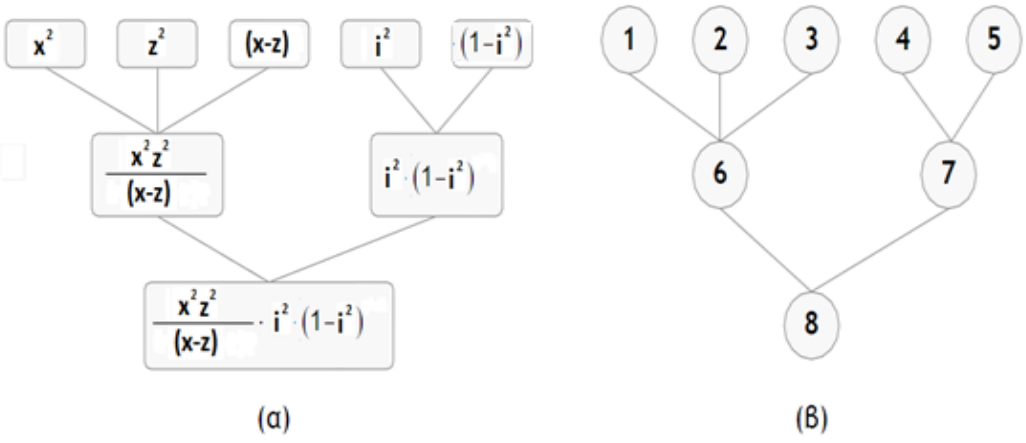
Κατά την έναρξη λειτουργιάς ενός TFlux προγράμματος αυτό διασπάται ,ούτος ώστε να αποτελείται μια σειρά από εντολές τα οποία ονομάζονται threads DDThr (data- driven threads)και τα οποία θα εκτελεστούν παράλληλα στην ίδια μηχανή που περιέχει πολλαπλούς πυρήνες .Οι εντολές αυτές έχουν την δυνατότητα να τρέχουν παράλληλα αφού είναι ανεξάρτητες μεταξύ τους αλλά και απαραίτητη προϋπόθεση ώστε να αρχίσει να εκτελείτε κάποιο thread είναι να είναι έτοιμα τα δεδομένα του που χρειάζεται .Υπεύθυνο για την ανάπτυξη εφαρμογής στην πλατφόρμα είναι το TFlux C Preprocessor το οποίο παίρνει ως είσοδο ένα κανονικό πρόγραμμα C συνδυασμένο με TFlux-ειδικές οδηγίες compiler (TFlux Directives) και παράγει τον παράλληλο κώδικα σε C για την αρχιτεκτονική του στόχου .Όλα τα Threads θεωρούνται πως είναι εξαρτημένα το ένα από το άλλο δηλαδή η εκτέλεση κάποιου

Thread μπορεί να προϋποθέτει την προηγούμενη εκτέλεση κάποιου άλλου .Για αυτό το λόγο κάθε Thread περιέχει ένα μετρητή (ReadyCounter) που συμβολίζει τα Threads από τα οποία περιμένει δεδομένα για να αρχίζει την εκτέλεση του . Σηματοδοτεί την έναρξη του Thread όταν αυτός γίνει ίσος με μηδέν. Για να γίνει ο καθορισμός αυτών των εξαρτήσεων χρησιμοποιούμε ένα γράφο συγχρονισμού ο όποιος διαγραμματικά αντιπροσωπεύει τις εξαρτήσεις μεταξύ των DThreads και επιπλέον, παρέχει πληροφορίες σχετικά με τον πυρήνα που εκτελεί κάθε DThread και το μπλοκ .Ένα μπλοκ περιέχει ένα αριθμό από Threads και ένας πυρήνας θεωρείται ως ένας επεξεργαστής του συστήματος.

Η πλατφόρμα TFlux περιέχει ένα preprocessor μέσο του οποίου γίνεται η παραγωγή του κώδικα αφού σαρώσει τα DDM προγράμματα .Έχει τον ρόλο του "παρατηρητή" ο οποίος έχοντας σαρώσει και αναλύσει το DDM πρόγραμμα θα συλλέξει τις πληροφορίες που δίνονται από το χρήστη ούτος ώστε να αρχίσει ο συγχρονισμός των Threads μέσο της μονάδας συγχρονισμού TSU (Thread Synchronization Unit) που θα δημιουργήσει το γράφο συγχρονισμού. Απαραίτητη θεωρείται μια αποδοτική επικοινωνία μεταξύ του TSU και της εφαρμογής πράγμα που επιτυγχάνεται με το runtime support .

Όπως έχει προαναφερθεί ένα thread για να είναι έτοιμο προς εκτέλεση πρέπει τα δεδομένα που χρειάζεται να είναι φορτωμένα στην μνήμη, έτοιμα. Αφού αυτά τα δεδομένα βρίσκονται στη μνήμη θα χρειαστεί να επικαλεστούμε αρκετές προσβάσεις στην μνήμη αφού έχουμε πολλαπλά thread .Αυτές οι προσβάσεις δεν γίνονται τυχαία αλλά καθορίζονται από πρωτόκολλο το οποίο φέρει το όνομα CacheFlow policy . Μέσο αυτής της πολιτικής επιλύονται και αρκετά προβλήματα που μπορεί να προκύψουν αφού έχει την δυνατότητα να ετοιμάζει τα δεδομένα στη μνήμη για τα threads πριν αυτά εκτελεστούν.

Ένα παράδειγμα του τρόπου λειτουργιάς του DDThr παρουσιάζεται πιο κάτω.



Σχήμα 2.2 Εκτέλεσης DDM Προγράμματος [14]

Όσο αφορά το συγκεκριμένο παράδειγμα το Thread 6 μπορεί να ξεκινήσει την εκτέλεση του μόνο αφού τα 1,2 και 3 ολοκληρώσουν. Αυτά που δεν έχουν καμία εξάρτηση μεταξύ τους μπορούν να εκτελεστούν παράλληλα. Όσο αφορά το ReadyCount το οποίο έχει ευθύνη να σηματοδοτεί την έναρξη του Thread η τιμή του για το 1,2,3,4 και 5 είναι ίση με μηδέν ,πράγμα που σημαίνει ότι είναι διαθέσιμα και έχουν την δυνατότητα να εκτελεστούν παράλληλα.
Το ReadyCount για το 6 έχει τιμή 3 αφού πρέπει να προηγηθούν τρεις εκτελέσεις Threads ενώ αντίστοιχα για το 7 είναι 2.Το ίδιο ισχύει και για το 8 δηλαδή 2 αφού αναμένει την ολοκλήρωση του 6 και 7.

2.5 Ροη εργασιών TFlux εργασίας

Τα TFlux προγράμματα αναπτύσσονται χρησιμοποιώντας ANSI-C μαζί με κάποια DDM directives . Τα DDM directives χρησιμοποιούνται για να εκφράσουν τον κώδικα των DDThrs, καθώς και τις μεταξύ τους εξαρτήσεις. Στη συνέχεια το πρόγραμμα περνά από τη σειρά εργαλείων μεταγλώττισης του TFlux και έτσι δημιουργείται το εκτελέσιμο αρχείο. Το πρώτο στάδιο αυτής της σειράς είναι ο TFlux Preprocessor, ο οποίος μεταφράζει το C πρόγραμμα με directives σε ένα αντίστοιχο παράλληλο C πρόγραμμα. Το δεύτερο στάδιο είναι η μεταγλώττιση του παραγόμενου παράλληλου C προγράμματος με ένα συνηθισμένο μεταγλωττιστή (gcc) και η παραγωγή ενός παράλληλου εκτελέσιμου αρχείου. Αυτό το εκτελέσιμο καλεί τις λειτουργίες του “TFlux Runtime Support”, το οποίο επιτρέπει την εκτέλεση με βάση το DDM. Το Runtime Support τρέχει πάνω από ένα Λειτουργικό Σύστημα τύπου Unix, το οποίο δε χρειάζεται να τροποποιηθεί κατά οποιοδήποτε τρόπο, και κρύβει όλες τις λεπτομέρειες σχετικά με το DDM, όπως τη συγκεκριμένη υλοποίηση του TSU. Μια από τις πρωταρχικές ευθύνες του Runtime Support είναι η δυναμική εγγραφή στα TSUs των μετα-δεδομένων (meta-data) των DDThrs και η κλήση όλων των TSU λειτουργιών που είναι απαραίτητες για την εκτέλεση

2.6 TFlux Directives

#pragma ddm thread T_ID kernel K_ID

Ορίζει το thread - νήμα στο οποίο θα εκτελούνται κάποιες διεργασίες. Η σταθερά τιμή T_ID δηλώνει τον αριθμό του νήματος και η τιμή K_ID των αριθμό των πυρήνων που θα εκτελεστεί το νήμα

#pragma ddm kernel 4

Η χρήση αυτής της οδηγίας γίνεται στην αρχή του προγραμματιστικού κώδικα και ορίζει των αριθμό των πυρήνων που θα εμπλέκονται στην εκτέλεση του προγράμματος.

#pragma ddm private var double x 4

Δήλωση private μεταβλητών που χρειάζονται για τοπικό σκοπό σε κάθε πυρήνα

#pragma ddm var int Y 4

Ορισμός Shared μεταβλητών για την ευρύ χρήση τους σε όλο το πρόγραμμα

#pragma ddm block blockID import (var varName1,…,var varNameN) export (varName1:ThreadIDx,…,varNameN:ThreadIDy)

ορισμός BLOCK που θα εμπεριέχονται αριθμός DThreads. Με το import εισάγουμε τις μεταβλητές προς χρήση στο block και με το export εξάγουμε της επιθυμητές μεταβλητές μετά από την χρήση τους σε κάποια Threads

#pragma ddm thread threadID kernel kernelID depends(ThreadID)

Καθορισμός των εξαρτήσεων μετάξι των DThreads.Για να αρχισει η εκτελεσεση του Thread προϋποθέτει την λήξη του Thread που ορίζουμε στο depends

DDM Loop
#pragma ddm for thread (t1, ... tn) kernel (k1, ... kn)
#pragma ddm endfor

Καθορίζει τον κώδικα στο σώμα του βρόχου, που θα εκτελείται μέσο των Threads T1 και TN στους πυρήνες K1 μέχρι Kn. Όλες οι επαναλήψεις του εν λόγω βρόχου είναι ανεξάρτητες

2.7 Παραδείγματα προγραμμάτων TFlux

Για την επίλυση κάποιου προγράμματος με το *TFlux* ,ο προγραμματιστής πρέπει να υλοποιήσει και να καθορίσει τις εξαρτήσεις μετάξι των Threads. Με αυτό τον τρόπο ένα Thread αν εξαρτάται από κάποιο άλλο τότε δεν μπορεί να αρχίσει την εκτέλεση του αν δεν ολοκληρώσει προηγουμένως το άλλο.

Στο παράδειγμα αυτό έχουμε τρεις μεταβλητές όπου θα γίνει ανάθεση τιμών . Η ανάθεση των τιμών γίνετε για κάθε μεταβλητή σε ξεχωριστό Thread 1,2 και 3 και τα οποία μπορούν να εκτελεστούν παράλληλα. Στο Thread 4 δηλώνεται ότι εξαρτάται (depends) από την προηγούμενη εκτέλεση των τριών ώστε να ξεκινήσει αυτό. Αν ανατεθούν οι τιμές και στα τρία τότε το $4^{ο}$ εκτελείται και δίνει αποτέλεσμα “Test Passed “ ενώ σε αντίθετη περίπτωση αν δεν εκτελεστούν σωστά βάση των εξαρτήσεων τα Threads τότε δίνει το αποτέλεσμα “Test Failed”. [6]

```
int main(int argc, char* argv[])
{
#pragma ddm kernel 3
#pragma ddm startprogram
#pragma ddm var int var01
#pragma ddm var int var02
#pragma ddm block 1
  #pragma ddm thread 1 kernel 1
    *var01 = 4;
#pragma ddm endthread
  #pragma ddm thread 2 kernel 2
```

```
    *var02 = 8;
#pragma ddm endthread
  #pragma ddm thread 3 kernel 3 depends(1, 2)
    if(*var01 == 4 && *var02 == 8)
     {
       printf("Test Passed");
     }
  #pragma ddm endthread
#pragma ddm endblock
#pragma ddm endprogram
return 0;}
```

Το Πρόγραμμα 2 που ακολουθεί περιλαμβάνει και δυο παράλληλους βρόχους όπου όλες οι επαναλήψεις των οποίων μπορούν να εκτελεστούν παράλληλα. Επισης χρησιμοποιείτε και η εντολή depends που δηλώνει την εξάρτηση των Threads.πρέπει να ολοκληρώσει την εκτέλεση του το πρώτο Thread για να ξεκινήσει το δεύτερο.

```
int main(int argc, char* argv[])

{
int i;
double sum = 0;
#pragma ddm kernel 8
#pragma ddm startprogram
#pragma ddm var double totalSum
*totalSum = 0;
#pragma ddm block 1
  #pragma ddm for thread 1
    for ( i = 0; i < 1024; i++ )
 {
A[i] = B[i] + C[i]
 }  #pragma ddm endfor
  #pragma ddm thread 2 kernel 1 depends (1)
#pragma ddm for thread 2
    for ( i = 0; i < 1024; i++ )
 {
Printf('%d" ,A[i] );
 }  #pragma ddm endfor

    printf("Result == %.2lf\n", *totalSum);
  #pragma ddm endthread
#pragma ddm endblock
#pragma ddm endprogram
return 0;
```

Κεφάλαιο 3^{ο} Το πλαίσιο MapReduce

3.1 Συστοιχίες υπολογιστών (Clusters)

Μια συστοιχία υπολογιστών (computer cluster) είναι δύο ή περισσότεροι υπολογιστές ,όχι αναγκαστικά ίδιου τύπου ή δυνατοτήτων, που συνδέονται μεταξύ τους ,συνήθως μέσω τοπικού δικτύου ή εικονικού ιδιωτικού δικτύου, προκειμένου να εκμεταλλευθούμε την δυνατότητα παράλληλης επεξεργασίας που αυτοί παρέχουν. Σήμερα υπάρχουν αρκετά διαφορετικά είδη συστοιχιών, κάθε ένα από τα οποία προσφέρει διαφορετικά πλεονεκτήματα στους χρήστες του.

3.2 Παραλληλισμός με MapReduce

Μία από τις επεξεργαστικές μονάδες οργανώνει τις υπόλοιπες για την εκτέλεση των υπολογισμών και φροντίζει να κατανέμει ισομερώς τον υπολογιστικό φόρτο, κρατώντας κατ' αυτό τον τρόπο συνεχώς απασχολημένες τις επεξεργαστικές μονάδες. Αυτή η τεχνική εφαρμόζεται μόνο σε συστήματα κατανεμημένης μνήμης, είναι αρκετά αποτελεσματική και η απόδοσή της εξαρτάται από τη διαθεσιμότητα της τοπικής μνήμης.

Για την λειτουργία του MapReduce , είναι απαραίτητη η αποθήκευση των δεδομένων εισόδου, αλλά και των αποτελεσμάτων, στο GFS (Google File System) ,το οποίο είναι ένα κατανεμημένο σύστημα αποθήκευσης εξαιρετικά μεγάλων δεδομένων σε πολλαπλές μηχανές, που ανήκουν σε ένα δίκτυο (cluster)

Η υλοποίηση αυτή ακολουθεί το μοντέλο αρχιτεκτονικής master/slave με ένα κεντρικό Master server «*jobtracker»* και πολλούς slave servers *«tasktrackers»,* ένα σε κάθε κόμβο του δικτύου μας.

3.3 Ορισμός του πλαισίου MapReduce

Το προγραμματιστικό μοντέλο MapReduce κυκλοφόρησε από την Google το 2003 και έχει ως σκοπός να διευκολύνει την επεξεργασία μεγάλων δεδομένων σε όγκο. Αυτή η νέα αφαίρεση σχεδιάστηκε για να μας επιτρέπει να εκφράζουμε τους απλούς υπολογισμούς της εκτέλεσης και κρύβει τις πολύπλοκες λεπτομέρειες του παραλληλισμού ,της ανοχής σφαλμάτων και της κατανομής των δεδομένων . Η υλοποίηση αυτή έχει εμπνευστεί από τα map και reduce primitives που παρουσιάστηκαν από την LIPS και άλλες συναρτησιακές γλώσσες . Ο χρήστης γράφει μόνο κώδικα για τον αλγοριθμικό μέρος ,και η υποδομή του MapReduce αναλαμβάνει την παράλληλη εκτέλεση, τον διαμοιρασμό των δεδομένων και τη διαχείριση των αποτυχιών. Γίνεται κατανομή της επεξεργασίας των δεδομένων σε πολλές υπολογιστικές μονάδες. Αυτό το κάνει επειδή εκμεταλλεύεται (large clusters) διασυνδεδεμένους υπολογιστές γενικής χρήσης (commodity hardware)και τους μοιράζει αντίγραφα μιας εργασίας ώστε να δουλεύουν παράλληλα για να την υλοποιήσουν .Οι προγραμματιστές βρίσκουν το σύστημα εύκολο στη χρήση, υπάρχουν περισσότερα από δεκάδες χιλιάδες προγράμματα που έχουν υλοποιηθεί από την Google τα τελευταία χρόνια και ένα μέσο όρο εκατό χιλιάδες MapReduce

εργασίες εκτελούνται καθημερινά και επεξεργάζονται συνολικά περισσότερα από είκοσι petabytes δεδομένων.

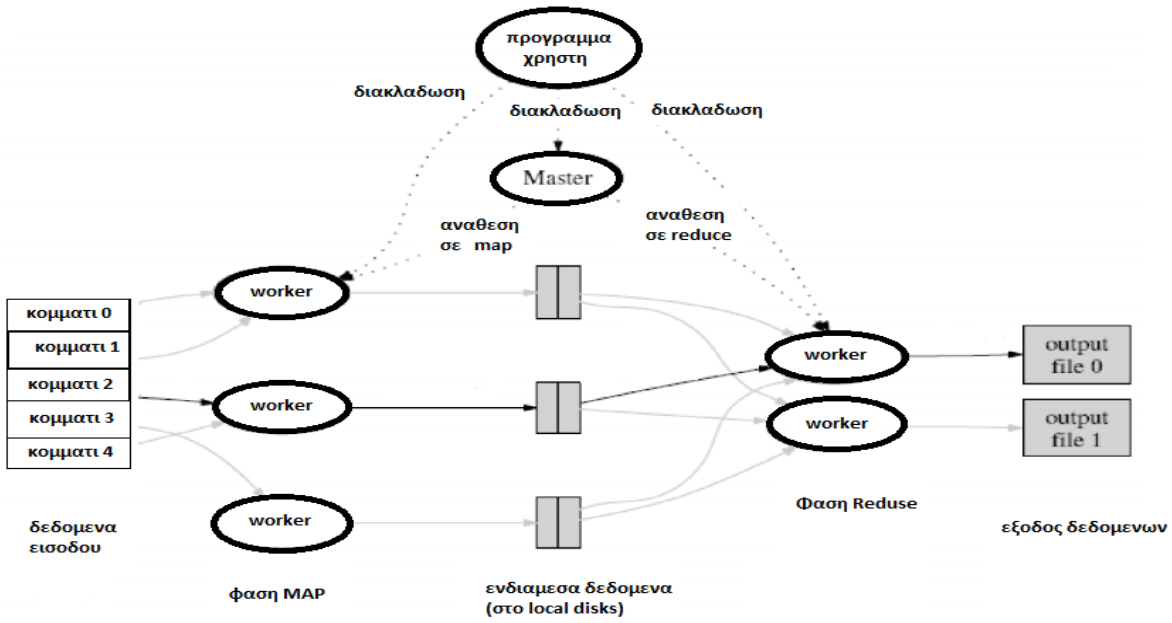
3.4 Λειτουργία του MapReduce

Το πλαίσιο λειτουργιάς της MapReduce αποτελείται από ένα σύνολο Map και Reduce εργασιών .Το μοντέλο δέχεται ως είσοδο ένα σύνολο ζευγών της μορφής κλειδί / τιμή . Το πλαίσιο χωρίζει τα δεδομένα σε μεγάλα ανεξάρτητα κομμάτια τα οποία υποβάλλονται σε επεξεργασία παράλληλα ,από εργασίες Map. Κάθε μια από αυτές τις εργασίες παράγει ένα σύνολο ενδιάμεσων ζευγών κλειδί /τιμή. Τα ζευγάρια αυτά ταξινομούνται με βάση τα ενδιάμεσα κλειδιά και περνούν στη συνάρτηση Reduce ,η οποία δέχεται ένα ενδιάμεσο κλειδί και ένα σύνολο τιμών που αντιστοιχούν σε αυτό το κλειδί .Οι τιμές συγχωνεύονται για να διαμορφώσουν ένα μικρότερο σύνολο τιμών.

3.5 Ροη των εργασιών MapReduce υπολογισμού

Τα αρχεία εισόδου διασπόνται σε M τμήματα των 16-64MB αλλά το ακριβός ποσό τον διαλέγει ο χρήστης .Ακολούθως ξεκινά στέλλοντας αντίγραφα του προγράμματος σε πολλαπλές μηχανές. Το ένα από αυτά τα αντίγραφα ονομάζεται master και τα άλλα workers γιατί σε αυτά δίδονται εργασίες από τον master .Συνολικά υπάρχουν M Map και P reduce εργασίες που πρέπει να δοθούν . Ειναι ευθύνη του master να διαλέξει workers που δεν είναι απασχολημένοι ώστε να τους δώσει μια Map και μια reduce εργασία. Ο worker που θα πιάσει την Map εργασία θα διαβάσει τα δεδομένα και θα τα αναλύσει σε ζευγάρια της μορφής κλειδί/τιμη και θα τα στείλει στην αντίστοιχη με το κλειδί συνάρτηση .Αυτή θα πιάσει τα ζευγάρια και θα παράξει στην έξοδο ενδιάμεσα ζευγάρια της μορφής κλειδί/τιμη που θα αποθηκεύσει στην μνήμη. Σε κάποια χρονικά διαστήματα αυτά τα αποθηκευμένα ζευγάρια θα γραφτούν στο τοπικό δίσκο και θα κατακερματιστούν σε R περιοχές-τοποθεσιες με την συνάρτηση κατακερματισμού . Ακολούθως οι τοποθεσίες των ζευγαριών θα σταλούν πίσω στον worker που αυτός με την σειρά του πρέπει να τα στείλει στους workers που προηγουμένως τους αναθέσαμε reduce εργασία. Στην συνέχεια ο reduce worker ειδοποιείτε από τον μάστερ για αυτές τις τοποθεσίες και θα κάνει ανάγνωση των αποθηκευμένων δεδομένων από τους τοπικούς δίσκους των Map workers . με το τελος της ανάγνωσης, τα ταξινομεί ώστε να δημιουργηθούν ομάδες που έχουν τις ίδιες εμφανίσεις κλειδιών. Αυτό επαναλαμβάνεται για όλους τους reduce workers μέχρι να ομαδοποιηθούν όλα με βάση τις εμφανίσεις των κλειδιών. Τέλος όταν ολοκληρωθεί κάθε Map και reduce εργασία ο μάστερ «ξυπνά» το πρόγραμμα του χρηστή.

Μια καλή αναπαράσταση των όσων έχουν λεχτεί πιο άνω για την ροη εργασιών μιας MapReduce εργασίας είναι το ακόλουθο



Σχήμα 3.1 Ροη των εργασιών MapReduce [15]

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

Είναι η υλοποίηση που έγινε για το προγραμματιστικό μοντέλο MapReduce πάνω σε shared memory συστήματα . 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.

3.6.1 Υλοποίηση Phoenix

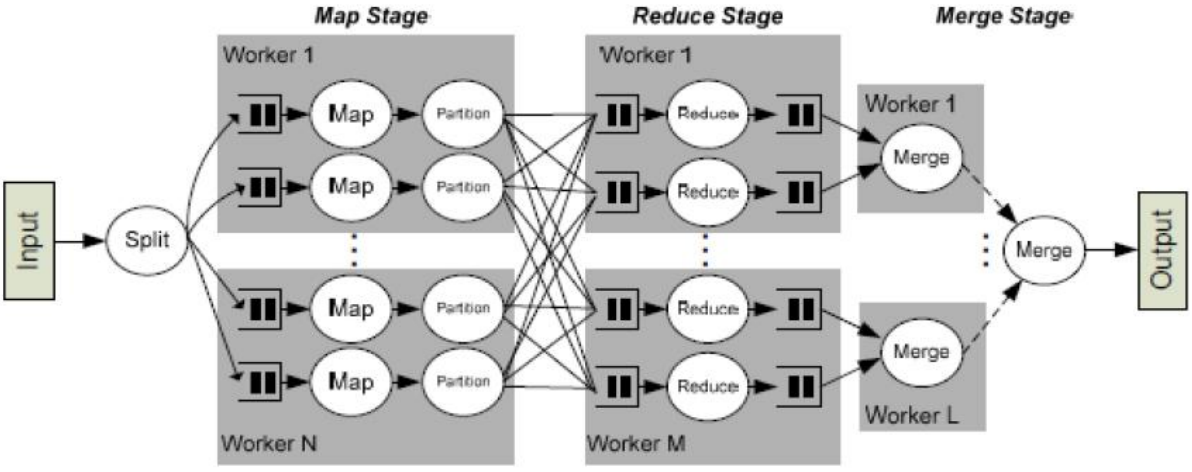
Όπως έχω ήδη αναφέρει στο κεφάλαιο το Phoenix είναι η υλοποίηση που έγινε για το προγραμματιστικό μοντέλο MapReduce πάνω σε συστήματα κοινόχρηστης μνήμης[2]. Περιλαμβάνει το programming API και το runtime system όπως και οι υπόλοιπες υλοποιήσεις που έχουν γίνει για το συγκεκριμένο προγραμματιστικό μοντέλο.

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.

3.6.2 Υλοποιημένες Εφαρμογές

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.

3.6.3 Οι συναρτήσεις του Phoenix API

Το Phoenix API δεν βασίζεται σε καμία συγκεκριμένη επιλογή από compiler και δεν απαιτεί μεταγλωττιστή παραλληλισμού. Ωστόσο, αυτό προϋποθέτει ότι οι λειτουργίες του μπορούν ελεύθερα να χρησιμοποιούν “stack-allocated and heap-allocated” δομές για προσωπικά “private” δεδομένα. Προϋποθέτει επίσης ότι δεν υπάρχει καμία επικοινωνία μέσω των δόμων της κοινής μνήμης, πλην των Buffers εισόδων / εξόδων για τις λειτουργίες αυτές.

Function Description	**R/O**
Functions Provided by Runtime	
`int phoenix_scheduler (scheduler_args_t * args)` Initializes the runtime system. The `scheduler_args_t` struct provides the needed function & data pointers	R
`void emit_intermediate(void *key, void *val, int key_size)` Used in Map to emit an intermediate output <key, value> pair. Required if the Reduce is defined	O
`void emit(void *key, void *val)` Used in Reduce to emit a final output pair	O
Functions Defined by User	
`int (*splitter_t)(void *, int, map_args_t *)` Splits the input data across Map tasks. The arguments are the input data pointer, the unit size for each task, and the input buffer pointer for each Map task	R
`void (*map_t)(map_args_t*)` The Map function. Each Map task executes this function on its input	R
`int (*partition_t)(int, void *, int)` Partitions intermediate pair for Reduce tasks based on their keys. The arguments are the number of Reduce tasks, a pointer to the keys, and a the size of the key. Phoenix provides a default partitioning function based on key hashing	O
`void (*reduce_t)(void *, void **, int)` The Reduce function. Each reduce task executes this on its input. The arguments are a pointer to a key, a pointer to the associated values, and value count. If not specified, Phoenix uses a default *identity* function	O
`int (*key_cmp_t)(const void *, const void*)` Function that compares two keys	R

Σχήμα 3.2 Οι λειτουργίες του Phoenix API. [8]

3.6.4 Παράδειγμα προγράμματος Phoenix MapReduce

wordCount υλοποίηση Phoenix-βασικές διαδικασίες [11]

map stage

```
void map(data_type const& s, map_container& out) const
{
    for (uint64_t i = 0; i < s.len; i++)
    {
        s.data[i] = toupper(s.data[i]);
    }

    uint64_t i = 0;
    while(i < s.len)
    {
        while(i < s.len && (s.data[i] < 'A' || s.data[i] > 'Z'))
            i++;
        uint64_t start = i;
        while(i < s.len && ((s.data[i] >= 'A' && s.data[i] <= 'Z') || s.data[i] == '\''))
            i++;
        if(i > start)
        {
            s.data[i] = 0;
            wc_word word = { s.data+start };
            emit_intermediate(out, word, 1);
        }
    }
}
```

Split stage

```
/** wordcount split()
 *  Memory map the file and divide file on a word border i.e. a space.
 */
int split(wc_string& out)
{
    /* End of data reached, return FALSE. */
    if ((uint64_t)splitter_pos >= data_size)
    {
        return 0;
    }

    /* Determine the nominal end point. */
    uint64_t end = std::min(splitter_pos + chunk_size, data_size);

    /* Move end point to next word break */
    while(end < data_size &&
        data[end] != ' ' && data[end] != '\t' &&
        data[end] != '\r' && data[end] != '\n')
        end++;

    /* Set the start of the next data. */
    out.data = data + splitter_pos;
    out.len = end - splitter_pos;

    splitter_pos = end;

    /* Return true since the out data is valid. */
    return 1;
}
```

sort stage

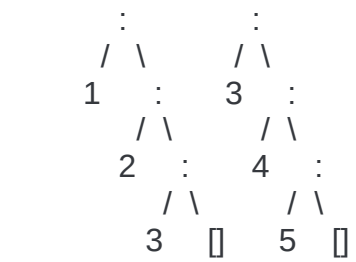
```
    bool sort(keyval const& a, keyval const& b) const
    {
        return a.val < b.val || (a.val == b.val && strcmp
(a.key.data, b.key.data) > 0);
    }
};
```

Κεφάλαιο 4^{ο} Mapping και Reducing(high order Functions)

4.1 Map/Reduce ως συναρτήσεις higher order

Η Map είναι higher order function η οποία λαμβάνει μια συνάρτηση και μια λίστα είτε πίνακα , και εφαρμόζει την εν λόγω συνάρτηση σε κάθε στοιχείο του πίνακα η λίστας και επιστρέφει την προκύπτουσα λίστα.

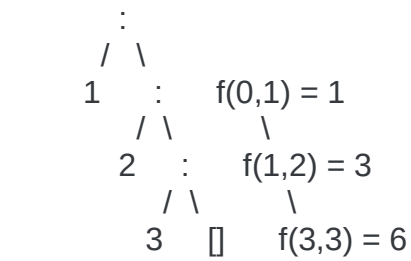
Παραδειγμα [7] αν περάσουμε σαν παράμετρο στη map την συνάρτηση x = x + 2, και μια λίστα από ακεραίους 1 -> 3. Η map θα επιστρέψει την εξής λίστα



Επιστρέφει: 3, 4, 5

Η Reduce παίρνει μια λίστα των στοιχείων και επιστρέφει μια συνάρτηση η οποία εφαρμόζεται σε όλα τα στοιχεία του καταλόγου αυτού.

Περνώντας της ως παράμετρο την συνάρτηση (x, y) => x + y, και την λίστα με τα στοιχειά 1 μέχρι 3 από το αποτέλεσμα της map , η Reduce θα επιστρέψει μια συνάθροιση του ακόλουθου



Επιστρέφει : 6

Χωρίς την κατανόηση του functional programming και των δυο higher order functions, δεν μπορούμε να κατανοήσουμε τον τρόπο λειτουργιάς του MapReduce, ο αλγόριθμος που κάνει το Google τόσο μαζικά επεκτάσιμο. Οι όροι map και reduce προέρχονται από Lisp και το συναρτησιακό προγραμματισμό

Για να γίνει η καλύτερη δυνατή κατανόηση του Functional programming εξετάζονται κάποια πολύ άπλα παραδείγματα

```
printf ("I'd like some Spaghetti!");

printf("I'd like some Chocolate Moose!");
```

Στην πραγματικότητα τα δυο αυτά μπλοκ κώδικα , είναι ακριβώς τα ίδια, εκτός από το ότι ένα μπλοκ αναφέρεται σε "σπαγγέτι" και το άλλο μπλοκ αναφέρεται σε "Moose σοκολάτα."

Ο επαναλαμβανόμενος κώδικας δεν φαίνεται σωστός ,έτσι δημιουργούμε μια συνάρτηση :

```
function Chef( food )
   {
      alert("I'd like some " + food + "!");
   }
   Chef("Spaghetti");
   Chef("Chocolate Moose");
```

Τώρα παρατηρούμε δύο άλλα τμήματα του κώδικα που μοιάζουν σχεδόν το ίδιο, εκτός από το ότι το ένα από αυτούς καλεί συνεχώς τη συνάρτηση BoomBoom και το άλλο καλεί συνεχώς τη συνάρτηση PutInPot. Εκτός από αυτά , ο υπόλοιπος κώδικας είναι λίγο πολύ ο ίδιος.

```
printf("get the lobster"); //τμήμα 1
   PutInPot("lobster");
   PutInPot("water");

  printf("get the chicken");//τμήμα 2
   BoomBoom("chicken");
   BoomBoom("coconut");
```

Τώρα χρειάζεται ένας τρόπος για να περάσει μια παράμετρος στη συνάρτηση που η ίδια η παράμετρος είναι μια συνάρτηση. Αυτή είναι μια σημαντική δυνατότητα, διότι αυξάνει τις πιθανότητες της εξεύρεσης κοινού κώδικα που μπορεί να βρίσκεται κρυμμένοι σε μια συνάρτηση.

```
function Cook( i1, i2, f )
   {
      printf("get the " + i1);
      f(i1);
      f(i2);
   }

   Cook( "lobster", "water", PutInPot );
   Cook( "chicken", "coconut", BoomBoom );
```

Έτσι παράμετρος στη συνάρτηση Cook είναι μια συνάρτηση f ,είτε η PutInPot είτε η BoomBoom

Ας υποθέσουμε ότι δεν έχουμε ορίσει ήδη τη συνάρτηση PutInPot ή BoomBoom. Δεν θα ήταν ωραίο αν μπορούσαμε απλά να τις ορίσουμε σε μια γραμμή αντί του πιο πάνω παραδείγματος

```
Cook( "lobster",        "water",     function(x)   { printf ("pot " + x); }  );
 Cook( "chicken",   "coconut",    function(x)    { printf ("boom " + x); } );
```

Παρατηρούμε ότι δημιουργούμε μια συνάρτηση χωρίς καν να την ονομάσουμε, απλά με το να την ορίσουμε και ταυτόχρονα να την περάσουμε σαν παράμετρο σε μια συνάρτηση.

Μόλις κατανοηθεί η χρήση ανώνυμων συναρτήσεις ως παράμετροι μπορούμε να εξετάσουμε συγκεκριμένα παραδείγματα , πχ παρατηρήσετε τον κωδικό, που κάνει κάτι σε κάθε στοιχείο ενός πίνακα.

```
var a = [1,2,3];
   for (i=0; i<a.length; i++)
   {
      a[i] = a[i] * 2;
   }
   for (i=0; i<a.length; i++)
   {
      printf(a[i]);
   }
```

Να επεξεργαζόμαστε κάθε στοιχείο με τον ίδιο τρόπο ενός πίνακα είναι αρκετά κοινό, και μπορούμε να γράψετε μια συνάρτηση που να κάνει αυτό για μας

```
function map(fn, a)
   {
      for (i = 0; i < a.length; i++)
      {
         a[i] = fn(a[i]);
      }
   }
```

Με όσα είδαμε προηγουμένως μπορούμε να γράψουμε αλλιώς την map

```
   map( function(x){return x*2;}, a );
   map( printf, a );
```

Ακόμη ένα κοινό πράγμα με τους πίνακες είναι η συγχώνευση “combine” όλων των τιμών του πινάκα με κάποιου τρόπο

```
function sum(a)
   {
      var s = 0;
      for (i = 0; i < a.length; i++)
         s += a[i];
      return s;
   }

   function join(a)
   {
      var s = "";
      for (i = 0; i < a.length; i++)
         s += a[i];
      return s;
   }
   printf(sum([1,2,3]));
   printf(join(["a","b","c"]));
```

Η συνάρτηση sum και join είναι τόσο ίδιες, γι αυτό καλό είναι να οργανωθούν σε μια γενική συνάρτηση η οποία συγχωνεύει τα στοιχειά του πινάκα σε μια απλή τιμή

```
function reduce(fn, a, init)
   {
      var s = init;
      for (i = 0; i < a.length; i++)
         s = fn( s, a[i] );
      return s;
   }

   function sum(a)
   {
      return reduce( function(a, b){ return a + b; },  a, 0 );
   }

   function join(a)
   {
      return reduce( function(a, b){ return a + b; },   a, "" );
   }
```

Πόσο όφελος μπορούμε πραγματικά να έχουμε από την δημιουργία μικρών συναρτήσεων που δεν κάνουν τίποτα περισσότερο από επαναλήψεις μέσα σε ένα πινάκα που κάνει κάτι σε κάθε του στοιχείο?

Προηγουμένως είχαμε δει τη συνάρτηση map και την οποία θα εξετάσουμε για περεταίρω ανάλυση . Όταν πρέπει να κάνουμε κάτι σε κάθε στοιχείο σε έναν πίνακα με τη σειρά , η αλήθεια είναι, πιθανώς δεν έχει σημασία με ποια σειρά θα τα κάνουμε. Μπορούμε να εκτελέσουμε μέσω του πίνακα προς τα εμπρός ή προς τα πίσω και να πάρουμε το ίδιο αποτέλεσμα. Στην πραγματικότητα, εάν έχουμε για παράδειγμα δύο επεξεργαστές, ίσως θα μπορούσαμε να γράψουμε κάποιο κώδικα που σε κάθε επεξεργαστή να κάνει τα μισά από τα στοιχεία, και ξαφνικά η map να είναι δύο φορές πιο γρήγορη στην εκτέλεση.

Αλλιώς , απλώς υποθετικά, έχουμε εκατοντάδες χιλιάδες servers σε διάφορα data centers σε όλο τον κόσμο, και έχουμε ένα πολύ μεγάλο πίνακα, που περιέχει υποθετικά, όλο το περιεχόμενο του Διαδικτύου. Τώρα μπορούμε να εκτελέσουμε την map σε χιλιάδες υπολογιστές, καθένας από τους οποίους θα αναλάβει ένα μικρό μέρος του προβλήματος.
Έτσι τώρα, για παράδειγμα, γράφοντας κάποιο πραγματικά γρήγορα κώδικα για να αναζητήσουμε όλα τα περιεχόμενα του Διαδικτύου είναι τόσο απλό όσο καλώντας τη συνάρτηση map με ένα βασικό string searcher ως παράμετρο.
Το πραγματικά ενδιαφέρον πράγμα που παρατηρούμε, εδώ, είναι ότι μόλις κατανοήσουμε την map και να reduce ως συναρτήσεις όπου ο καθένας μπορεί να χρησιμοποιήσει, δεν έχουμε παρά να γράψουμε τον υπόλοιπο κώδικα για να τρέξει την map και reduce σε μια παγκόσμια μαζική παράλληλη συστοιχία ηλεκτρονικών υπολογιστών clusters , και να λειτουργεί χιλιάδες φορές πιο γρήγορα το οποίο σημαίνει ότι μπορεί να χρησιμοποιηθεί για την αντιμετώπιση τεράστιων προβλημάτων .

4.2 Χρήση των Map /Reduce στο MapReduce της Google

Το MapReduce είναι μια συγχώνευση από δυο higher order functions που λαμβάνονται από το λειτουργικό προγραμματισμό “functional programming” τις map και Reduce. Χρησιμοποιήθηκε αρχικά για να δημιουργηθεί ένα αντεστραμμένο ευρετήριο του διαδικτύου. «inverted index». Είναι ένα framework της Google που χρησιμοποιείται για την επεξεργασία 20 petabytes δεδομένων την ημέρα. Το MapReduce παρά το γεγονός ότι είναι γραμμένο σε C + +, η φιλοσοφία πίσω από MapReduce είναι Functional, εμπνευσμένη από το map και reduce primitives που παρουσιάζονται στο Lisp και σε πολλές άλλες functional languages .

Αυτό που επιτυγχάνεται μέσο του Συναρτησιακού Προγραμματισμού “Functional Programming” είναι ότι μια συνάρτηση μπορεί να χρησιμοποιηθεί ως παράμετρος εισόδου ή τιμή επιστροφής για ορισμό μιας συνάρτησης «Higher Order Function».

Ανάμεσα στις διάφορες Higher Order Functions, είναι και η map, fold και filter που είναι και οι πιο δημοφιλείς:
- Map είναι μια Higher Order Function που εφαρμόζει μια συγκεκριμένη λειτουργία (γνωστός και ως μετασχηματιστής). Ο Μετασχηματιστής είναι μια συνάρτηση που

εφαρμόζεται σε κάθε στοιχείο μιας λίστας και θα παράγει ένα ή περισσότερα νέα στοιχεία.

map (toLower) "! abcDEFG12 @ #" θα παράγει έξοδο: "! abcdefg12 @ #"

- fold (γνωστή ως Reduce) είναι μια Higher Order Function που επεξεργάζεται (χρήση μιας συνάρτησης συνδυαστή combiner) τον κατάλογο των στοιχείων σε κάποια κατάταξη για να δημιουργήσει μια τιμή επιστροφής. Ο combiner είναι μια συνάρτηση που εφαρμόζεται σε δύο στοιχεία και παράγει ένα αποτέλεσμα που μπορεί να συνδυαστεί με το combiner μαζί με τα υπόλοιπα στοιχεία της λίστας. Για παράδειγμα: reduce (+) 0 [1 .. 5] θα παράγει εξόδου: 15, που είναι το άθροισμα όλων των στοιχείων.

Ουσιαστικά, αυτές οι δυο higher order functions εφαρμόζουν μια λειτουργίας σε κάποια λίστα / πίνακα και παράγουν κάποια αποτελέσματα: map μετατρέπει κάθε στοιχείο και reduce συνενώνει όλα τα στοιχεία.Το μοντέλο αυτό της Google ονομάζεται MapReduce, αλλά οι έννοια τους είναι κάπως διαφορετική
Η σημασία της map είναι η ίδια όπως και στην functional programming language, ο μετασχηματιστής (ο mapper στο έγγραφο της Google), εφαρμόζεται σε κάθε στοιχείο της λίστας
- Η σημασία του reduce διαφέρει. Εδώ, ο combiner (reducer), εφαρμόζεται σε πολλα επιμέρους τμήματα από τα στοιχεία της λίστας και επομένως παράγει πολλαπλά reduce αποτελέσματα, αλλά στη συναρτησιακή γλώσσα προγραμματισμού εφαρμόζεται σε όλα τα στοιχεία και παράγει μόνο ένα αποτέλεσμα.
Εννοιολογικά το πώς τα στοιχεία χωρίζονται σε πολλαπλά επιμέρους τμήματα;
Για να επιλύσουμε αυτό το πρόβλημα, το μοντέλο αυτό παρουσιάζει κάποια δομή για τα στοιχεία που παράγονται από mapper και καταναλώνονται από reducer - κάθε στοιχείο / αρχείο έχει δύο μέρη “το κλειδί και τιμή”. Στη συνέχεια, όλα τα στοιχεία χωρίζονται σύμφωνα με την κλείδα. Τα στοιχεια με την ίδια τιμή από ενα επιμέρους τμήμα στο σύνολό της περνουν σε ένα reducer .

Από την πλευρά της εφαρμογής του, το πιο σημαντικό πλεονέκτημα του Προγραμματισμού μοντέλου αυτού είναι ότι - επιτρέπει την αυτόματη παραλληλοποίηση και διανομή της επεξεργασίας μιας μεγάλης κλίμακας στοιχείων:
- Mapper εφαρμόζεται σε κάθε αρχείο , όπου και πρόκειται για μια παράλληλη διαδικασία χειρισμού δεδομένων από μόνη της , εμείς απλά πρέπει να διανέμουν τα δεδομένα εισόδου στο όριο μεταξύ των κόμβων επεξεργασίας.
- ο reducer εφαρμόζεται σε κάποιο επιμέρους τμήμα, εμείς απλά πρέπει να διανεμηθούν τα επιμέρους τμήματα μεταξύ των κόμβων επεξεργασίας.

Κεφάλαιο 5^{ο} Σχετικές εργασίες – υλοποιήσεις MapReduce

5.1 Παραλληλισμός Map/Reduce υπολογισμού με OpenMP

Το προγραμματιστικό μοντέλο του OpenMP είναι βασισμένο σε παραλληλισμό με νήματα. Μια διεργασία κοινής μνήμης αποτελείται από πολλά νήματα και το OpenMP βασίζεται στο χαρακτηριστικό αυτό. Στο OpenMP ο προγραμματιστήςμπορεί να ελέγχει και να αυξάνει ή να μειώνει τον αριθμό των νημάτων Ένα νήμα αρχηγός (master) εκτελεί τις εντολές σειριακά μέχρι να φτάσει την πρώτη παράλληλη περιοχή δημιουργώντας καινούργιες ομάδες από νήματα όταν χρειάζεται. Η ομάδα των νημάτων που περιλαμβάνεται σε μια παράλληλη περιοχή εκτελείται παράλληλα. Όταν η ομάδα τελειώσει την εργασία της, τότε τα νήματα συγχρονίζονται και τερματίζουν, αφήνοντας μόνο το νήμα αρχηγό και η διαδικασία αυτή επαναλαμ-βάνεται όσες φορές χρειάζεται.

Για να γίνει η παραλληλοποίηση, ο προγραμματιστής θα πρέπει να δώσει στον μεταφραστής οδηγίες οι οποίες ενσωματώνονται στον κώδικα της C. Στη συνέχεια, ο μεταφραστής μεταφράζει αυτές τις οδηγίες και παράγει κλήσεις σε συναρτήσεις βιβλιοθήκης έτσι ώστε να παραλληλοποιήσει τμήματα του κώδικα.

Παράδειγμα μμετατροπής σειριακού προγράμματος C σε πρόγραμμα με χρήση C με οδηγία OpenMP

```
void main()
{
  double x[1000];
  for(int i = 0; i < 1000; i++)
  {
    big_calc(x[i]);
  }
}
```

```
void main()
{
  double x[1000];
#pragma omp parallel for
  for(int i = 0; i < 1000; i++)
  {
    big_calc(x[i]);
  }
}
```

Σχήμα 5.1 παράδειγμα χρήσης OpenMP [20]

Οι αλλαγές που έγιναν ήταν οι:

• προσθήκη του αρχείου δηλώσεων omp.h στην κορυφή του κώδικα και

• προσθήκη ενός #pragma omp parallel for ακριβώς πριν από την επανάληψη.

Όταν ο προεπεξεργαστής του μεταφραστή συναντήσει την οδηγία αυτή, αναλαμβάνει αυτόματα να παράγει κώδικα τέτοιο που θα αναλάβει τον καταμερισμό του βρόχου επανάληψης σε τμήματα εργασίας Τα τμήματα αυτά θα ανατεθούν σε μια ομάδα νημάτων για εκτέλεση, μέχρι να εκτελεστούν όλα. Μετά το τέλος του βρόχου

επανάληψης που ακολουθεί την οδηγία #pragma, το πρόγραμμα θα εκτελεστεί ως σειριακό και κατά τον τερματισμό του θα καταστραφεί και η ομάδα νημάτων που δημιουργήθηκε. Ο τελικός κώδικας που μεταφράζεται περιέχει όλες τις απαραίτητες δομές και δηλώσεις και το εκτελέσιμο πρόγραμμα εκτελείται πολυνηματικά.

Μπορούμε να χρησιμοποιήσουμε OpenMP για παραλληλησουμε ένα στυλ map/reduce υπολογισμό. Ένα πρόβλημα που θα λύσουμε θα ειναι ο υπολογισμός της συνολικής ενέργειας αλληλεπίδρασης μεταξύ κάθε ιόντων σε ενα πινακα από ιόντα και ενος reference_ion . Σε μια συνάρτηση υπολογισμού θα περναμε το ιον στο οποιο αναφερόμαστε reference_ion, καθώς και το πίνακα από ιόντα. Ο αλγόριθμος που θα εκτελείται για κάθε ιόν του πίνακα θα είναι αναφέρεται στον υπολογισμό της απόσταση μεταξύ των ιόντων του πίνακα και το ιόν αναφοράς.

Χρησιμοποιήστε αυτή την απόσταση (r) για τον υπολογισμό της ενέργειας αλληλεπίδρασης (1 / r)

Προσθέστε αυτή την ενέργεια αλληλεπίδρασης πάνω στο συνολικό ποσό.

Map/reduce μπορεί να χρησιμοποιηθεί όταν έχουμε ένα πίνακα από δεδομένα, μια συνάρτηση που επιθυμούμε να εφαρμόσουμε (να κάνουμε mapping) σε κάθε στοιχείο του πίνακα, και μια ενιαία τιμή που θέλουμε πίσω που είναι η μείωση των αποτελεσμάτων της εφαρμογής της συνάρτησης σε κάθε στοιχείο του πίνακα. Όσον αφορά τους ορους map/reduce, ο αλγόριθμος μας θα μοιάζει με αυτό:

Δημιουργήστε μια συνάρτηση η οποία υπολογίζει και επιστρέφει την αλληλεπίδραση μεταξύ του ιόν που περνάμε σε αυτή και του ιόν αναφοράς , π.χ. calc_energy (ion)

Map κάθε ιόν στον πίνακα κατά τη energy συνάρτηση calc_energy
Μειώστε/ reduce το αποτέλεσμα της κάθε mapped συνάρτησης χρησιμοποιώντας ένα sum (+)

Υλοποίησης κώδικα[18]:

```
#include <stdio.h>
#include <math.h>

/* Define an Ion type to hold the  coordinates of an Ion */
typedef struct Ion
{
   double x;
   double y;
   double z;
} Ion;

/* The reference Ion */
struct Ion reference_ion;

/* Return the square of a number */
double square(double x)
```

```
{
   return x*x;  }

/* Energy function to be mapped */
double calc_energy( struct Ion ion )
{
   double r;

   r = sqrt( square( reference_ion.x - ion.x ) +
            square( reference_ion.y - ion.y ) +
            square( reference_ion.z - ion.z ) );

   /* The energy is simply 1 / r */
   return 1.0 / r;
}

/* You will need to fill in this function to read in
   an array of ions and return the array */
struct Ion* readArrayOfIons(int *num_ions)
{
   int i;
   Ion *ions = (Ion*)malloc(10 * sizeof(Ion));

   *num_ions = 10;

   for (i=0; i<10; ++i)
   {
      ions[i].x = 0.0;
      ions[i].y = 0.0;
      ions[i].z = i;
   }

   return ions;
}

int main(int argc, char **argv)
{
   int i, num_ions;
   struct Ion *ions_array;
   double total_energy, mapped_energy;

   /* Lets put the reference ion at (1,2,3) */
   reference_ion.x = 1.0;
   reference_ion.y = 2.0;
   reference_ion.z = 3.0;

   /* Now read in an array of ions */
   ions_array = readArrayOfIons( &num_ions );

   total_energy = 0.0;
```

```
    #pragma omp parallel private(mapped_energy) \
                        reduction( + : total_energy )
    {
        mapped_energy = 0.0;

        #pragma omp for
        for (i=0; i < num_ions; ++i)
        {
            /* Map this ion against the function */
            mapped_energy += calc_energy( ions_array[i] );
        }

        /* Reduce to get the result */
        total_energy += mapped_energy;
    }

    printf("The total energy is %f\n", total_energy);

    return 0;
}
```

5.2 MapReduce για την αρχιτεκτονική Cell B.E

Για να εφαρμοστεί το συγκεκριμένο MapReduce προγραμματιστικό μοντέλο στην αρχιτεκτονική του επεξεργαστή Cell χρειάζεται να πάρει τα χαρακτηριστικά του μοντέλου. Όταν ο χρήστης 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.

Η αρχιτεκτονική του Cell B.E. είναι πολυεπεξεργαστική και περιλαμβάνει δύο ειδών επεξεργαστικά στοιχεία. Αυτά είναι ένας επεξεργαστής αρχιτεκτονικής Power και οχτώ επεξεργαστικά στοιχεία που λειτουργούν όλα με συχνότητα ρολογιού 3.2GHz. Η επεξεργαστική μονάδα αρχιτεκτονικής Power (PPE, Power Processing Element) υποστηρίζει δύο ταυτόχρονα hardware-threads και διαθέτει κρυφές μνήμες πρώτου επιπέδου ξεχωριστές για εντολές και δεδομένα και μια κοινή δευτέρου επιπέδου. Τα οκτώ ακόμη επεξεργαστικά στοιχεία ονομάζονται συνεργατικά (SPE, Synergistic Processing Element) και είναι αυτά που παρέχουν το σημαντικότερο μέρος της υπολογιστικής ισχύος του Cell.

Σύμφωνα με το runtime system του Cell , οι map και οι reduce εργασίες θα εκτελεστούν μόνο από τους SPEs χωρίς να εμπλακεί 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

Κατά το **στάδιο map**, οι SPEs εκτελούν τη συνάρτηση map, που 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. Τα δεδομένα εισέρχονται στην τοπική μνήμη των SPEs αφού το runtime system τρέχει στον PPE για να αρχικοποιήσει την εκτέλεση αυτής της φάσης Τα δεδομένα εισόδου διαιρούνται σε 8 κομμάτια ιδίου μεγέθους, ένα κομμάτι για κάθε SPE όπου σε κάθε

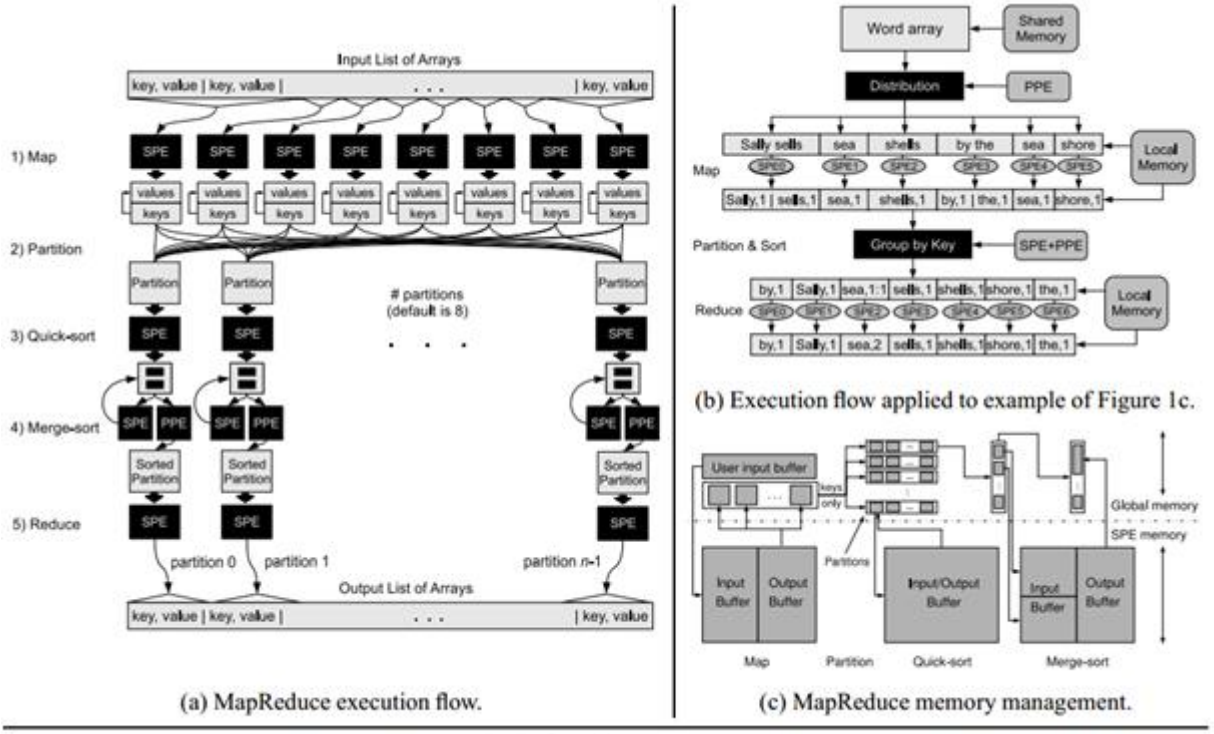
ένα δεσμεύεται και ένα buffer για να αποθηκεύονται τα δεδομένα εξόδου .Όταν γεμίσει μεταφέρονται στο αντίστοιχο buffer στη κοινόχρηστη μνήμη.

Στο **στάδιο Partition** η hashing τα ζεύγη με το ίδιο κλειδί ομαδοποιούνται. Ο PPE θα ψάξει τα αποτελέσματα των SPEs δηλαδή κάθε κλειδί και να το τοποθετήσει στη ανάλογη ομάδα.

Στο στάδιο **Quick-sort** οι ομάδες δεδομένων που δημιουργήθηκαν από το partition θα μεταφερθούν από την κοινόχρηστη μνήμη στους SPEs, θα ταξινομηθούν και μετά θα επιστραφούν ξανά στην κοινόχρηστη μνήμη.

Στο **στάδιο Merge Sort** θα δοθούν στους SPEs τα sorted buffers που έγιναν μέσο του quick sort στους οποίους θα εκτελεστεί το merge-sorting γιατί υπάρχει περίπτωση να μην έγινε σωστή ταξινόμηση λόγο της μικρής τοπικής μνήμης των SPEs

Στο στάδιο Reduce εκτελείτε η από τη χρήστη ορισμένη συνάρτηση reduce 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.



Σχήμα 5.2 MapReduce for the Cell B.E. Architecture[6]

Κεφάλαιο 6^{ο} Παραλληλισμός Map/Reduce υπολογισμού με TFlux

6.1 Mapping Reducing με TFlux

Όπως και στην σχετικές εργασίες που έχουν αναφερθεί ακλουθούσε μια τυπική μεθοδολογία παραλληλισμού ενός MapReduce υπολογισμού ,υλοποιώντας ουσιαστικά τους Mappers τους Reducers με αντίστοιχη διαδικασία του χρήστη και παραλληλισμού τους με Threads, έτσι και στο TFlux θα κρατηθεί η ίδια μεθοδολογία.

Παρόμοια με την μεθοδολογία που χρησιμοποιήθηκε η υλοποίηση βασίζεται σε 3 βασικά στάδια, τα οποία βοηθούν στην εκτέλεση μιας εφαρμογής, τα στάδια map, διαδικασία δυαδικής αναζήτησης και reduce.

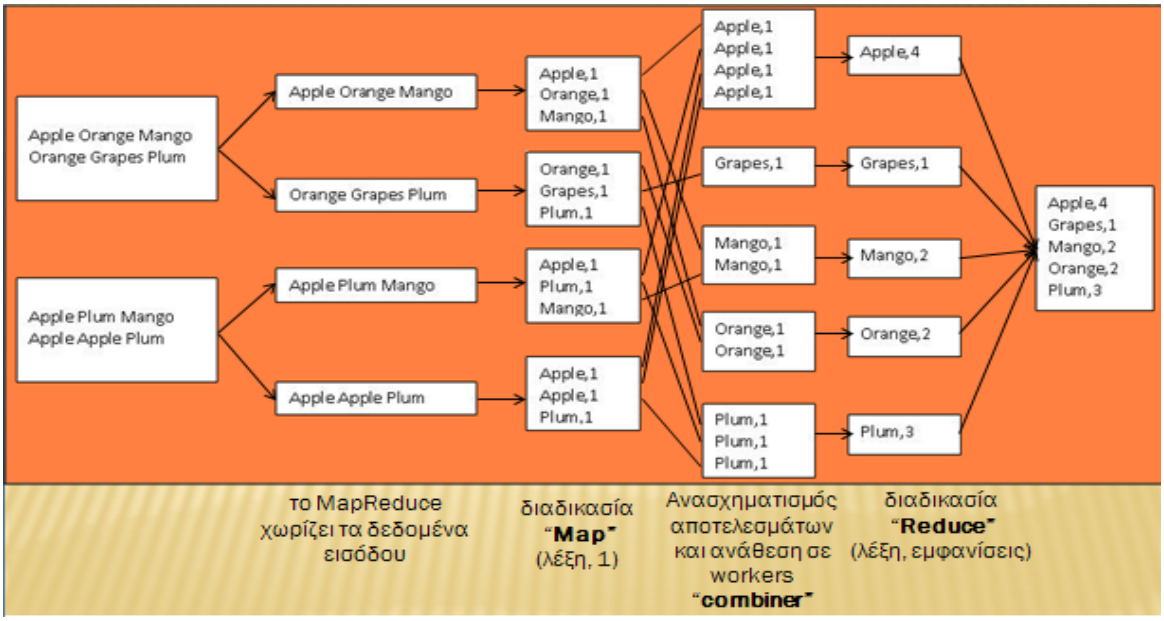
Κατά το στάδιο **map**, οι πυρήνες εκτελούν τη συνάρτηση map, που ορίζει ο χρήστης πάνω στα δεδομένα εισόδου Τα δεδομένα εισόδου διαιρούνται σε n κομμάτια ιδίου μεγέθους, ένα κομμάτι για κάθε πυρήνα της μηχανής όπου σε κάθε ένα δεσμεύεται και ένα buffer για να αποθηκεύονται τα δεδομένα εξόδου .Όταν γεμίσει μεταφέρονται στο αντίστοιχο buffer στη κοινόχρηστη μνήμη.

Στο στάδιο της **δυαδικής αναζήτησης** τα αποτελέσματα τα οποία προκύπτουν από τους mappers θα ομαδοποιηθούν ,σε λογαριθμικό χρόνο, βάση κοινών χαρακτηριστικών ,παράδειγμα την ίδια τιμή /value.

Στο στάδιο **Reduce** θα δοθούν στους πυρήνες τα ομαδοποιημένα αποτελέσματα που έγιναν μέσο της δυαδικής αναζήτησης στους οποίους θα εκτελεστεί η διαδικασία Reduce όπως ορίζεται από το χρήστη για κάθε πυρήνα, και θα γραφτεί το τελικό αποτέλεσμα της συγχώνευση των τιμών.

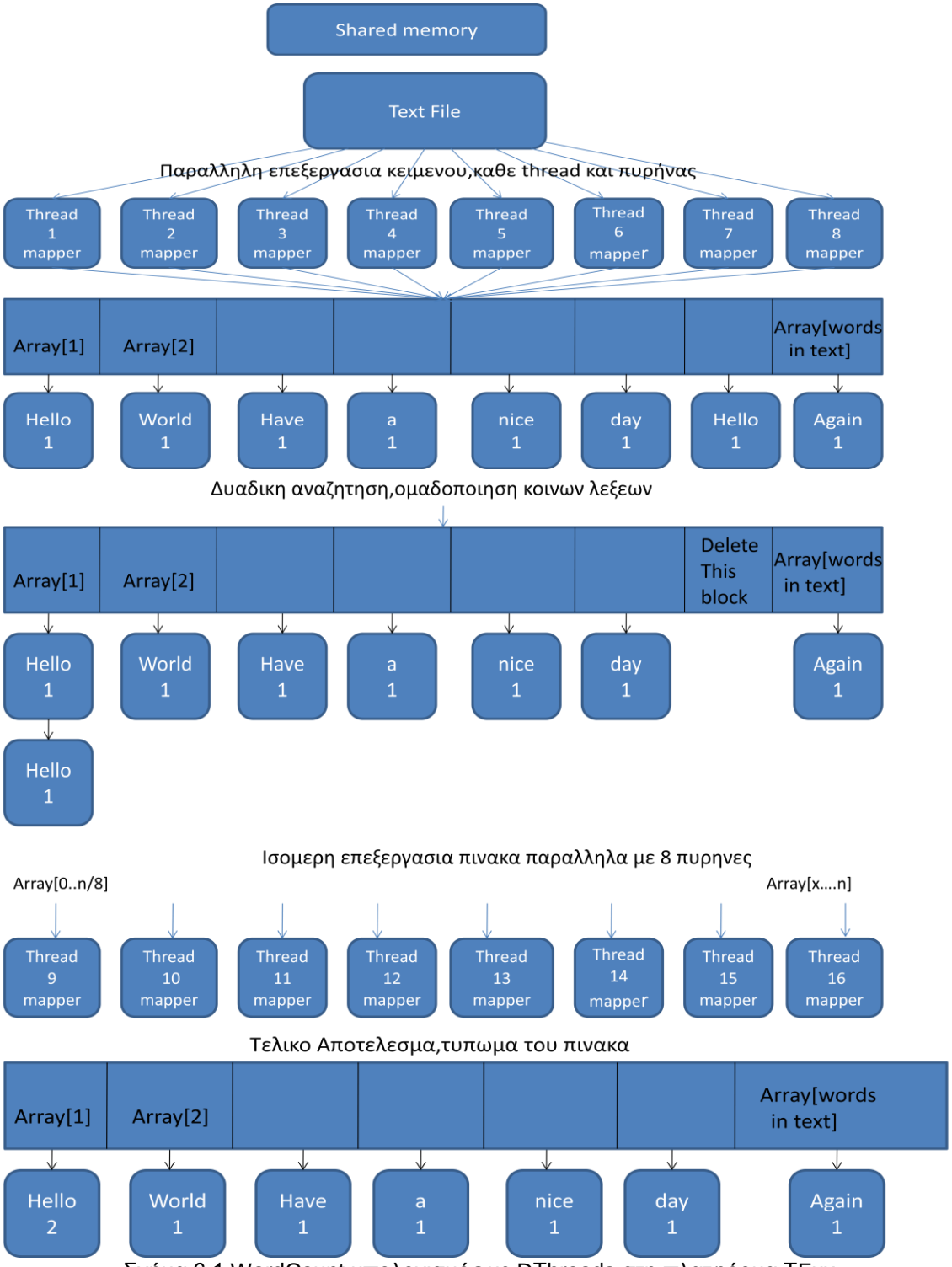
6.2 Παραλληλισμός WordCount MapReduce με TFlux

Ούτος ώστε να γίνει κατανοητό ο τρόπος με τον οποίο μπορεί να γίνει αυτό εφικτό θα εφαρμοστεί η μεθοδολογία αυτή σε κάποια παραδείγματα προγραμμάτων του MapReduce . Ένα από αυτά είναι το **WordCount**, εφαρμογή κατά την οποία υπολογίζεται η συχνότητα εμφάνισης κάθε λέξης σε ένα σύνολο αρχείων .



Σχήμα 6.1 παράδειγμα εκτέλεσης MapReduce wordCount

Αφού η μεθοδολογία έχει διατυπωθεί μπορούμε να χρησιμοποιήσουμε το παράδειγμα του wordCount ώστε να δηχθεί παραστατικά και να διερευνηθεί η υλοποίηση της μεθοδολογίας αυτής. Ακολουθεί η αναπαράσταση της εκτέλεσης του wordCount παραστατικά.



Σχήμα 6.1 WordCount υπολογισμός με DThreads στη πλατφόρμα TFux

Κατά την ανάλυση του πιο πάνω διαγράμματος διαφαίνονται κάποια βήματα κλιμάκωσης, μεθοδολογία όπως είπαμε και πριν που χρησιμοποιήθηκε και έχει εξής.

Κατά το 1ο ΒΗΜΑ από την κοινή μνήμη εισάγουμε στους mappers (όσοι και οι πυρήνες της μηχανής που ενδεχομένως να χρησιμοποιηθεί)ώστε να διαβάζουν παράλληλα το κείμενο γραμμή –γραμμή και ακολούθως να σπάσουν τις γραμμές σε λέξεις και να τις αποθηκεύσουν σε πίνακα λιστών .Μέσο της διαδικασίας mapping δημιουργείται για κάθε λέξη του κειμένου, συναρτησιακά , καινούργιος κόμβος του πίνακα λιστών όπου θα αποθηκευτεί η λέξη και θα της ανατεθεί τιμή ίση με 1, το αντίστοιχο δηλαδή emit (key,value). Όλοι οι mappers εκτελούνται σε ένα block που προϋποθέτει την λήξη του για να συνεχιστεί η υπόλοιπη διαδικασία

Ακολούθως στο 2ο Βημα μέσο της διαδικασίας της δυαδικής αναζήτησης (αντίστοιχο Merge Sort στο Phoenix) ομαδοποιούμε το πίνακα με της κοινές λέξεις ,δηλαδή οι λέξεις με τις ίδιες εμφανίσεις

Στο 3^{ο} Βημα οι λίστες που προκύπτουν περνούν σε reducers (όσοι και οι πυρήνες της μηχανής που ενδεχομένως να χρησιμοποιηθεί)και αθροίζονται ώστε να έχουμε αποτέλεσμα (λέξη , εμφανίσεις). Και τέλος στο 4^{ο} Βήμα τυπώνουμε το πίνακα λιστών που προκύπτει μετά τις αλλαγές στις οποίες έχει υποστεί από τους Reducers και παρουσιάζουμε τα αποτελέσματα της διαδικασίας

6.2.2 Ψευδοκωδικας WordCount με TFlux

////////////////////////////////////**mapping διαδικασία** ////////////////////////////

/*έναρξη του πρώτου Block που συμπεριλαμβάνει τους Mappers */

#pragma ddm Block 1

/*οορισμός του πρώτου Thread και της διαδικασίας mapping που εκτελείται σε αυτό*/

#pragma ddm thread 1 kernel 1

/* διαβάζουμε τις λέξεις από το αρχείο ανά γραμμή με fgets*/

```
while (fgets(line1,MAX_STR_LEN,fp1) != NULL)
 *(strchr(line1,'\n')) = '\0';
```

/*διασπούμε την γραμμή σε λέξεις */

```
for (s1 = strtok(line1,WS); s1 != NULL; s1 = strtok(NULL,WS)) {
```

/*για κάθε λέξη δημιουργούμε ορίζουμε καινούργια θέση του πίνακα δυναμικά*/

```
 array = (int*) malloc(1*sizeof(int));
```

/*κάθε θέση του πίνακα ορίζεται ως μια λίστα επίσης δυναμικά */

```
        struct listNode *array [i] = (struct listNode*)calloc( 1,sizeof( struct listNode ) );
/*εισάγεται η λέξη στη λίστα με αρχική τιμή 1 //emit(word,value)*/
        insertWord(array[i],s1);i++;
/*αύξηση μετρητή όσες και οι λέξεις του κειμένου*/
              count1 ++;
/*Τέλος  του Thread και της διαδικασιας mapping*/
#pragma ddm endthread
```

/////ανάθεση Mapping διαδικασίας σε Threads αντίστοιχους δηλαδή Mappers/////

```
/*ορίζουμε των αριθμό πυρήνων προς χρήση*/
#pragma ddm kernel 8
#pragma ddm thread 1 kernel 1
/*mapping διαδικασια.....
.....*/
#pragma ddm endthread
/*Καθε Mapper εκτελειται σε διαφορετικο πυρηνα (kernel) παραλληλα*/
#pragma ddm thread 2 kernel 2
/*mapping διαδικασια.....
.....*/
#pragma ddm endthread
/*το ιδιο μοτιβο για n πυρηνες οσους και  της μηχανης */
```

//////////////////διαδικασια δυαδικής αναζήτησης κοινών λέξεων//////////////////////

```
Binary search(struct listNode *array)
/*ελέγχουμε ανά δυο τις θέσεις του πίνακα
for(i = 0; i < count-1; i++) {
   for(j = 0; j < count-i-1; j++)    {
     if (a[j+1]->word == a[j]->word)…..}
```

/*με την εύρεση κοινής λέξης δημιουργούμε καινούργιο κόμβο στη λίστα*/

struct listNode q = newListNode(s);

while ((array[j]->p->next != NULL)

/*ο κόμβος θα προστεθεί στη θέση του πίνακα με την κοινή λέξη δηλαδή αυξάνεται η λίστα για την συγκεκριμένη λέξη σε αριθμό όσο και οι εμφανίσεις τις */ p = p->next;

p->next = q;

  q->word = p->word

q->value=1;

/*αποδεσμεύουμε την κοινή λέξη*/

free array[j+1];

/*μειώνουμε κατά ένα το μέγεθος του πίνακα*/

count--; }

////////////////////////////////////Reducing διαδικασία//

/*Επεξεργαζόμαστε τον καινούργιο πίνακα .Κάθε reducer αναλαμβάνει ορισμένα στοιχειά αυτού που ο αριθμός τους ορίζεται με βάση το διαθέσιμο αριθμό πυρήνων της μηχανής δηλαδή ο πινάκας λιστών διασπάται εικονικά σε ισομερή υπο πίνακες για την παράλληλη επεξεργασία του*/

int x=count mod num Kernels;

z=0;

/*περνούμε σαν παράμετρο το πίνακα και θέσεις του πίνακα όπου θα επεξεργαστεί ο reducer αρχίζοντας από την x μέχρι την z

#pragma ddm thread 9 kernel 1

Reducer(array[struct listNode *array],x,z)

/*για κάθε στοιχείο του πίνακα από x μέχρι z*/

for(i = x; i < =z ; i++) {

/*για κάθε στοιχείο του πίνακα που αντιπροσωπεύει μια λέξη διασχίζουμε την λίστα του */

while ((array[j]->p->next != NULL)

/*για κάθε κόμβο της λίστας του αυξάνουμε ένα μετρητή που είναι ουσιαστικά και οι εμφανίσεις κάθε λέξης*/

occurance++;

/*αποδεσμεύουμε τον κόμβο που διασχίσαμε */

free (array[i]->p)

temp=array->word;

/*όταν διανύσουμε όλους τους κόμβους και αποδεσμευτούν τότε έχουμε το occurance= εμφανίσεις λέξης και απλά προσθέτουμε ξανά για την συγκεκριμένη θέση του πίνακα ένα κόμβο λίστας με την λέξη και τις εμφανίσεις */

array[i]->word=temp;

array[i]->value=occurance;

#pragma ddm endthread

//ανάθεση Reducing διαδικασίας σε Threads αντίστοιχους δηλαδή Reducers ////

#pragma ddm thread 9 kernel 1

z=x; /*αρχίζει η διαδικασία από την θέση που θα τελειώσει άλλος reducer*/

x+=x; /*μέχρι να επεξεργαστεί τον ίδιο αριθμό στοιχείων*/

/*Reducing διαδικασία..... Reducer(array [struct listNode *array] , x , z)

.....*/

#pragma ddm endthread

/*Καθε Reducer εκτελείται σε διαφορετικό πυρήνα (kernel) παράλληλα*/

/*μπορούμε και πάλι να χρησιμοποιήσουμε kernel από τους προηγούμενους με την λήξη των mapper */

#pragma ddm thread 10 kernel 2

/*Reducinging διαδικασία.....

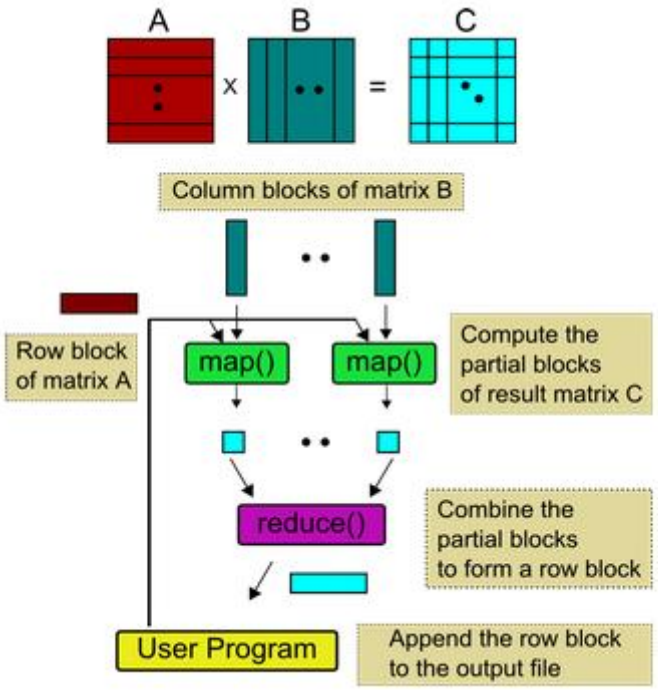
.....*/

#pragma ddm endthread

/*το ίδιο μοτίβο για n πυρήνες όσους και της μηχανής */

6.3 Παραλληλισμός Matrix Multiply MapReduce με TFlux

Ένα επιπρόσθετο παράδειγμα παραλληλισμού MapReduce υπολογισμού με TFlux που μελετηθεί είναι αυτό του Matrix Multiply που υλοποιεί αλγόριθμο για πολλαπλασιασμό πινάκων ακεραίων αριθμών. Ο αλγόριθμος αυτός βασίζεται στην στρατηγική επεξεργασίας διασπασμένων στηλών/γραμμων των πινάκων. Θεωρούμε ένα παράδειγμα όπου σαν είσοδο δεχόμαστε δυο πινάκες A και B και παράγουν ένα τρίτο C ως αποτέλεσμα του πολλαπλασιασμού των πινάκων. Διασπούμε το πίνακα B σε ένα μπλοκ από στήλες και το πίνακα A σε ένα μπλοκ από γραμμές. Σε κάθε επανάληψη όλες οι map διεργασίες επεξεργάζονται δυο εισόδους (i) ένα μπλοκ από στήλες του B (ii) ένα μπλοκ από γραμμές του A πίνακα .Από αυτές παράγεται ένα μπλοκ από γραμμές του τελικού πίνακα C . Κάθε μπλοκ από στήλες αντιστοιχίζεται σε κάθε map διεργασία ενώ κάθε μπλοκ από γραμμές αλλάζει σε αυτές σε κάθε επανάληψη .



Σχήμα 6.1 MapReduce αλγόριθμος για Matrix Multiplication [8]

Για την υλοποίηση του αλγορίθμου αυτού με TFlux θα γίνει χρήση μιας σημαντικής TFlux directive η οποία έχει να κάνει με την επίτευξη παράλληλης εκτέλεσης των νημάτων σε ένα βρόγχο.

```
DDM Loop
#pragma ddm for thread ( t1, ... tn ) kernel ( k1, ... kn )
#pragma ddm endfor
```

Καθορίζει τον κώδικα στο σώμα του βρόχου, που θα εκτελείται μέσο των Threads T1 και TN στους πυρήνες K1 μέχρι Kn. Όλες οι επαναλήψεις του εν λόγω βρόχου είναι ανεξάρτητες

Έχοντας την δυνατότητα χρήσης μια τέτοιας εντολής μας απαλλάσσει από πολλές σκέψεις ως προς την υλοποίηση του MapReduce matrix multiply με Tflux Directives. Αυτό αρχικά που αξίζει να παρατηρηθεί είναι ότι δεν μπορούμε να εφαρμόσουμε την ίδια διαδικασία που γίνεται στους mappers. Στο MapReduce σε κάθε επανάληψη όλες οι map διεργασίες επεξεργάζονται ένα μπλοκ από στήλες του B και ένα μπλοκ από γραμμές του A πίνακα. Κάθε μπλοκ από στήλες αντιστοιχίζεται σε κάθε map διεργασία. Στο TFlux βασική επιδίωξη είναι να εκμεταλλευτούμε όσο το δυνατό τους προσφερόμενους πυρήνες της μηχανής ώστε να τρέχουμε διεργασίες σε αυτούς παράλληλα .Οπόταν είναι αδύνατο για παράδειγμα σε ένα πινάκα 400χ400 να δεσμεύσουμε 400 πυρήνες να εκτελεστούν οι map διεργασίες.

Άρα αυτό που μπορούμε να κάνουμε είναι να εκτελούμε την διαδικασία πολλαπλασιασμού στηλών/γραμμών που εφαρμόζεται στο πολλαπλασιασμό πινάκων ,παράλληλα στους διαθέσιμους πυρήνες χωρίς ουσιαστικά να διασπούμε τους πινάκες σε μπλοκ γραμμών/στηλων .Έτσι απλά οι πίνακες θα επεξεργάζονται ως έχει παράλληλα από πολλαπλά Threads να αναθέτονται σε πυρήνες με την πιο πάνω TFlux Directive .Για να γίνει πιο κατανοητή η χρήση του '*pragma ddm for'* δίνετε παράδειγμα εφαρμογής του.

```
#pragma ddm for thread 1
  for ( i = 0; i < 1024; i++ )
  {
A[i] = B[i] + C[i]
  }
#pragma ddm endfor      //όλοι οι πυρήνες συμμετέχουν στο βρόγχο(ο παραλληλισμός αυξάνεται αυτόματα με τους πυρήνες)
```

Επίσης μπορούμε με μια μικρή προσθήκη της εντολής "unroll" να επιφέρουμε λιγότερες επαναλήψεις του βρόγχου τεχνική loop unrolling (ξεδίπλωμα βρόχου) είναι μια από τις πιο διαδεδομένες τεχνικές βελτιστοποίησης κώδικα. Στην τεχνική αυτή, διαδοχικές επαναλήψεις του αρχικού βρόχου γράφονται σαν ξεχωριστές εντολές

```
#pragma ddm for thread <number> unroll <number>
for(i = 0; i < 1024; i++)
A[i] = i;
for(i = 0; i < 256; i+=4)
{
A[i] = i;
A[i+1] = i+1;
A[i+2] = i+2;
A[i+3] = i+3;
} #pragma ddm endfor
```

//το number είναι μια σταθερά που η τιμή της εξαρτάται από το χρήστη και δηλώνει τον επιθυμητό αριθμό ξεδιπλώματος του βρόγχου.

Τώρα μπορούμε την γνώση αυτή να την εφαρμόσουμε στο matrix multiply για παραλληλισμό του MapReduce υπολογισμού με TFlux

6.3.1 Ψευδοκωδικας matrix multiply με TFlux

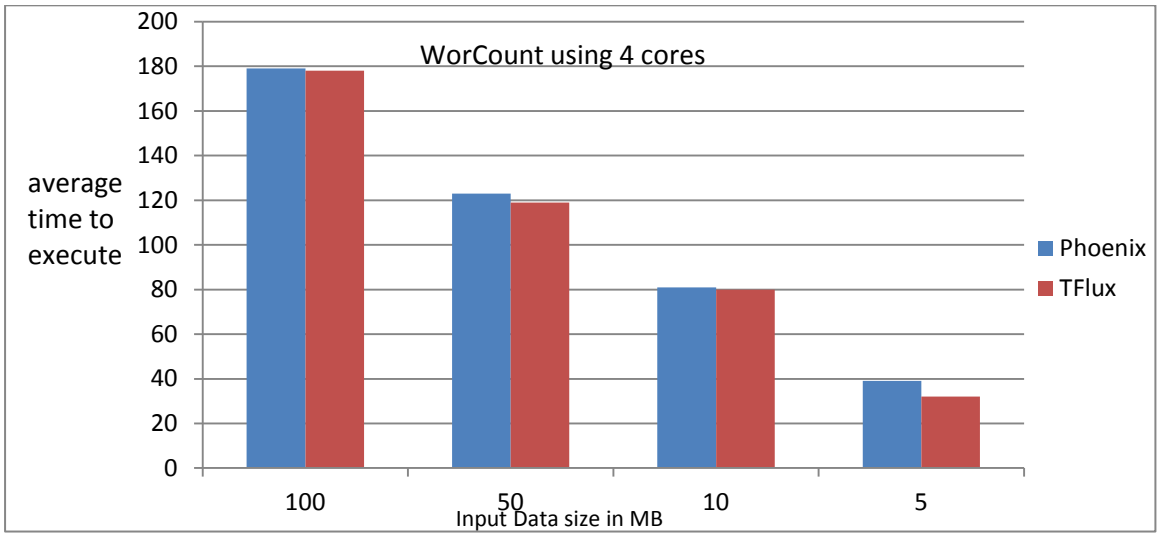
```
/*δεχόμαστε ως δεδομένα εισόδου δυο πίνακες*/

float a[size][size];

float b[size][size];

/*κατασκευάζουμε ένα τρίτο πινάκα που θα είναι και ο πίνακας πολλαπλασιασμού τους */

float c[size][size];

int main()

{

   // Map συνάρτηση…Compute matrix multiplication.   C = C + A x B

//μπορούμε να εφαρμόσουμε unroll όπου θεωρείται ευνοϊκό

        #pragma ddm for thread 1
      for (int j = 0; j < size; ++j) {

         for (int k = 0; k < size; ++k) {

            c[i][j] += a[i][k] * b[k][j];

       #pragma ddm endfor   }
 }

//Reduce συνάρτηση….προαιρετική

//μπορούμε να αθροίσουμε τις τιμές του πολλαπλασιαστικού πινάκα C και σαν έξοδο να έχουμε μια τιμή .Εκτελείται αυτόματος παράλληλα όπως και η map

        #pragma ddm for thread 2

      for (int j = 0; j < size; ++j) {

         for (int k = 0; k < size; ++k) {

            sum +=c[j][k] ;

                }
```

7 Πειράματα- Απόδοση

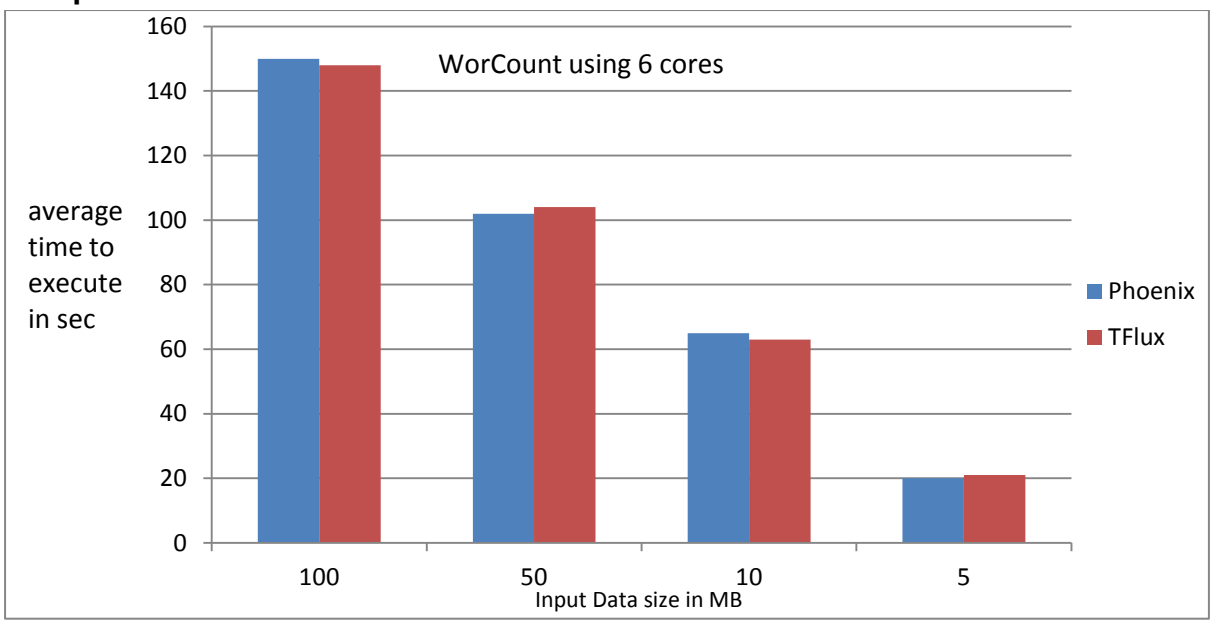
7.1.1 Εκτέλεση παραδειγμάτων(μετρήσεις)

Έχουν παρθεί μετρήσεις που αφορούν το χρόνο εκτέλεσης από τις εφαρμογές matrix multiply και word count. Συγκεκριμένα έτρεξε κάθε εφαρμογή χρησιμοποιώντας διαφορετικό αριθμό πυρήνων από το σύστημα, αρχίζοντας από 4 πυρήνες, συνεχίζοντας με 6 και τέλος με 8 πυρήνες καταγράφοντας κάθε φορά τον χρόνο σε δευτερόλεπτα που απαιτητέ προς ολοκλήρωση της εκτέλεσης τους . Ακολούθως, ανάλογα με τον αριθμό των πυρήνων που χρησιμοποιήθηκαν, χρειάστηκε να υπολογιστεί ο μέσος όρος των χρόνων για διαφορετικό όγκο δεδομένων .Συγκεκριμένα για το WordCount χρησιμοποιήθηκαν αρχεία μεγέθους 5MB, 10MB, 50MB και 100MB.Αντοιστοιχα για το matrix multiply χρησιμοποιήθηκαν πίνακες μεγέθους 4000 γραμμών και στηλών ,2000,1000 και 500^{ων}.

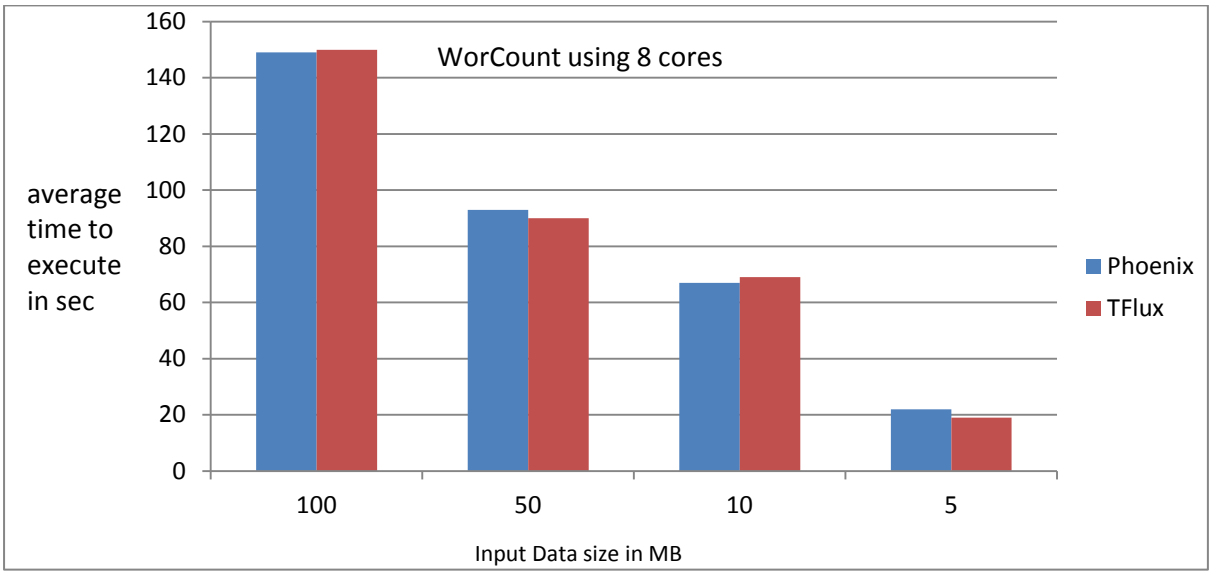
Σενάριο 1^{ο}



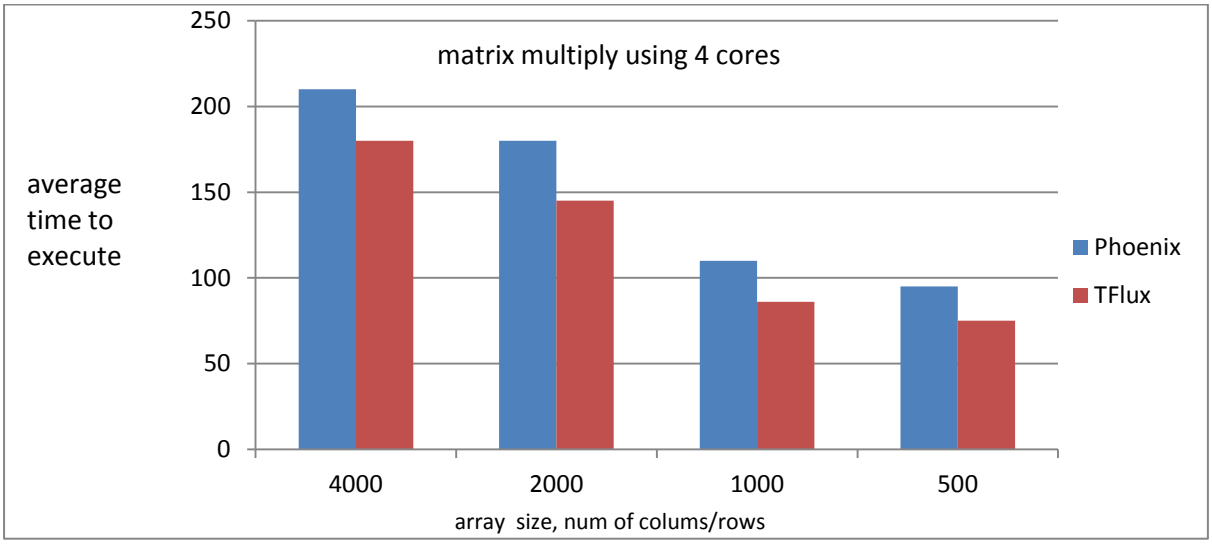
Σενάριο 2^{ο}



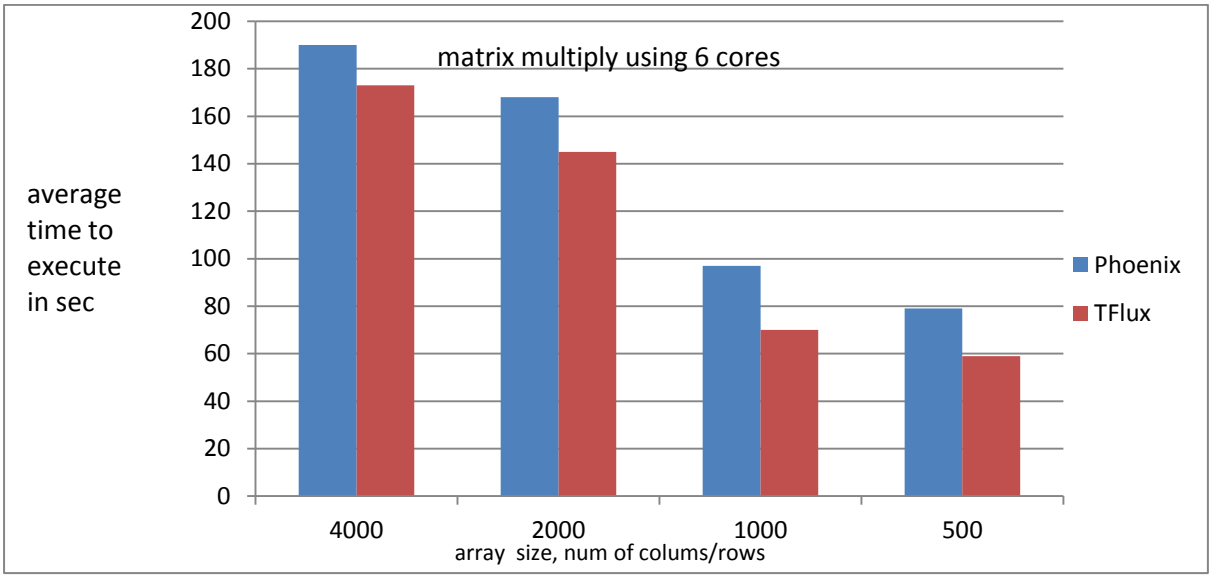
Σενάριο 3ο



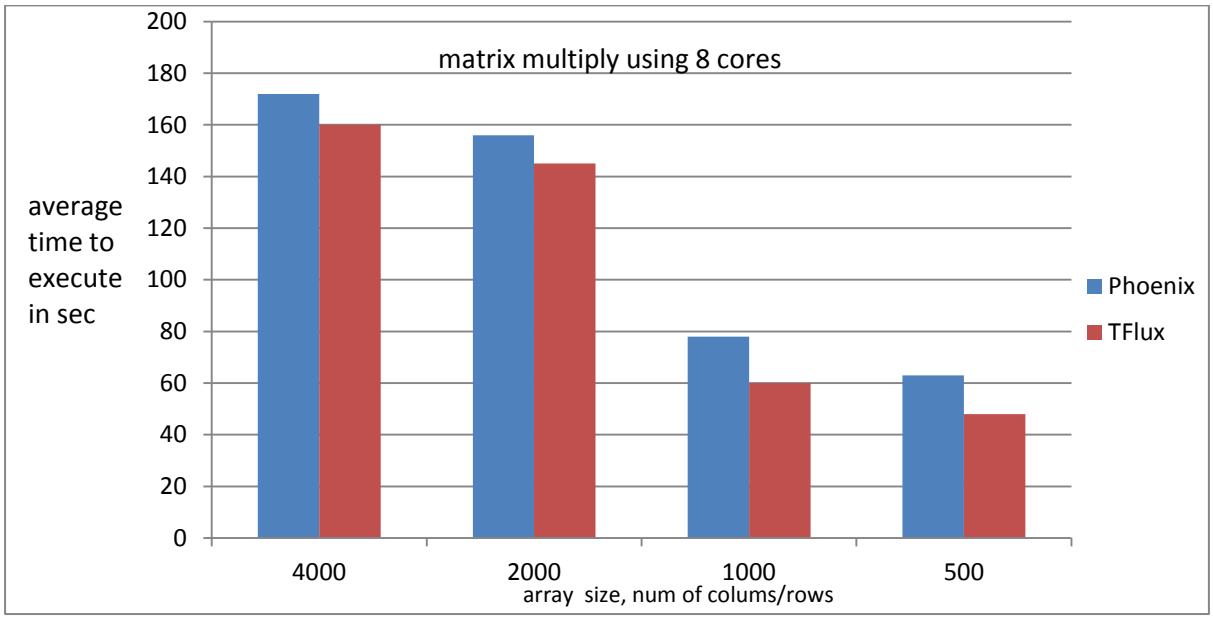
Σενάριο 4ο



Σενάριο 5ο



Σενάριο 6^{ο}



7.1.2 Παρατηρήσεις από Πειράματα

Με αυτά τα σενάρια μπορεί τώρα να εξαρθούν κάποια συμπεράσματα όσο αφορά την απόδοση των εφαρμογών TFlux υλοποίησης με αυτό του Phoenix MapReduce . Από τη γραφικές παραστάσεις παρατηρείται ότι η διαφορά στο χρόνο εκτέλεσης των Phoenix και TFlux δεν διαφέρουν και πολύ. Όσο αφορά ειδικά την εφαρμογή WordCount η διάφορα είναι πάρα πολύ μικρή με το TFlux να διαφαίνεται ότι είναι ελαφρά ταχύτερο σε εκτέλεση απότι το Phoenix. Το ίδιο παρατηρείται και για την εφαρμογή matrix multiply με τις επιδόσεις να διαφέρουν ακόμη πιο πολύ και το TFlux να φαίνεται αρκετά αποδοτικό.

7.2 Πλεονεκτήματα TFlux

Το Flux Thread (TFlux), είναι ένα ολοκληρωμένο σύστημα που υποστηρίζει το μοντέλο εκτέλεσης Data-Driven Multithreading (DDM). Περιλαμβάνει ένα επεξεργαστή ο οποίος μαζί με ένα σύνολο απλών οδηγιών pragma, επιτρέπει στο χρήστη να *αναπτύξει εύκολα* DDM προγράμματα. Αυτό του δίνει αρκετά πλεονεκτήματα.

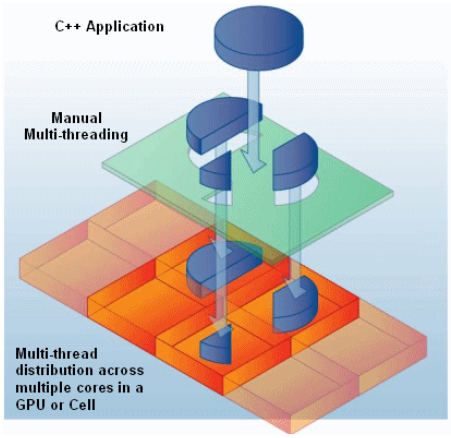
Το Data-Driven Multithreading (DDM) είναι ένα μοντέλο εκτέλεσης παραλληλισμού που έχει την προέλευσή του στο Dynamic Dataflow . Η υιοθέτηση των τεχνικών του Dynamic Dataflow έχουν ένα πλεονέκτημα, καθώς παρέχει το μέγιστο διαθέσιμο παραλληλισμό Ένα τέτοιο λειτουργικό εκμεταλλεύεται στο έπακρο τα πολυεπεξεργαστικα συστήματα.

Μέσα σε ένα DThread οι οδηγίες εκτελούνται κατά ένα τρόπο ελέγχου ροής, αξιοποιώντας τυχόν βελτιστοποιήσεις που προσφέρει ο επεξεργαστής εκτέλεσης .

Εκτός από το σημαντικό όφελος στις επιδόσεις, το DDM μοντέλο διαθέτει επίσης ένα πλεονέκτημα σε επίπεδο προγραμματισμού, αφού για τα περισσότερα προγράμματα είναι σχετικά εύκολο να εντοπίσει τον κώδικα του DThreads, καθώς και τα στοιχεία συσχέτισης μεταξύ τους.

Ο προεπεξεργαστής παράγει αυτόματα τον TFlux κώδικα, ο οποίος μπορεί να μεταγλωττιστεί με τη χρήση οποιουδήποτε εμπορικού C compiler, επομένως αυτομάτως γίνεται παραγωγή κώδικα σε οποιαδήποτε ISA. Το TFlux προσφέρει ένα προεπεξεργαστή που επιτρέπει στις εφαρμογές εύκολα να μεταφερθούν σε TFlux απλά αυξάνοντας συνήθεις C κώδικες με τις κατάλληλες οδηγίες pragma

Επιπρόσθετα με τις χρήσεις νημάτων Threads επιτυγχάνεται η παράλληλη εκτέλεση πολλών νημάτων δηλαδή αξιοποίηση χωριστών νήματων για κλήσεις εισόδου/ εξόδου και ταυτόχρονη επικοινωνία με χρήστη και επεξεργασία. Ακόμη γίνεται η αξιοποίηση εγγενούς παραλληλισμού της εφαρμογής με τη χρήση όλων των διαθέσιμων επεξεργαστών και τη γρήγορη επικοινωνία μέσω κοινής μνήμης



Σχήμα 7.4 Χειρισμός multithreading [20]

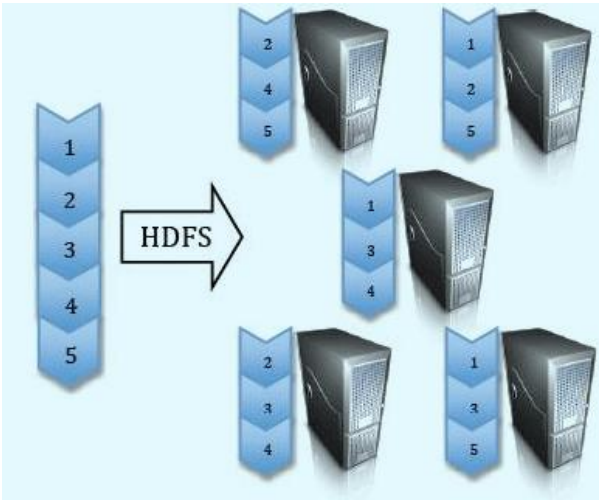
7.6 Πλεονεκτήματα Hadoop MapReduce

Το Hadoop είναι προσπελάσιμο , τρέχει σε μεγάλα clusters από εμπορικές μηχανές ή σε υπηρεσίες cloud computing όπως το Elastic Compute Cloud(EC2) της Amazon.

Είναι ανθεκτικό γιατί προορίζεται να τρέχει σε εμπορικό υλικό, το Hadoop έχει δομηθεί με την υπόθεση των συχνών δυσλειτουργιών του υλικού. Μπορεί εύκολα να χειριστεί τέτοιες αποτυχίες.

Το Hadoop κλιμακώνεται γραμμικά για να διαχειριστεί περισσότερα δεδομένα προσθέτοντας περισσότερους κόμβους στο cluster.

Επίσης είναι αξιόπιστο και σε περίπτωση βλάβης γίνεται αυτόματη ανάθεση των εργασιών σε νέους κόμβους. Επίσης παρέχει τη δυνατότητα αυτόματης αποθήκευσης των δεδομένων σε πολλαπλά αντίγραφα.



Σχήμα 7.5 Hadoop HDFS διαδικασία [3]

7.7 Σημαντική διαφορά των δυο

Η Εξισορρόπηση φορτίου είναι μία από τις πιο δύσκολες πτυχές του παράλληλου προγραμματισμού και την καθιστά πολύ πιο δύσκολη με τη χρήση λογισμικών που λειτουργούν με κατανεμημένες μνήμες όπως το εν λόγο MapReduce. Αντίθετα ,όπως το TFlux, σε κοινόχρηστη μνήμη μπορούμε κατανέμουμε τα καθήκοντα για αποδοτική επικοινωνία μεταξύ των πόρων και συνάμα ισάξια κατανομή εργασιών σε αυτούς. Αν και η εξισορρόπηση φορτίου δεν είναι τόσο σημαντική στην εκτέλεση ενός αλγορίθμου MapReduce, καθίσταται απαραίτητη κατά το χειρισμό μεγάλων αρχείων, για την επεξεργασία του υλικού και όταν η χρήση των πόρων είναι ζωτικής σημασίας.

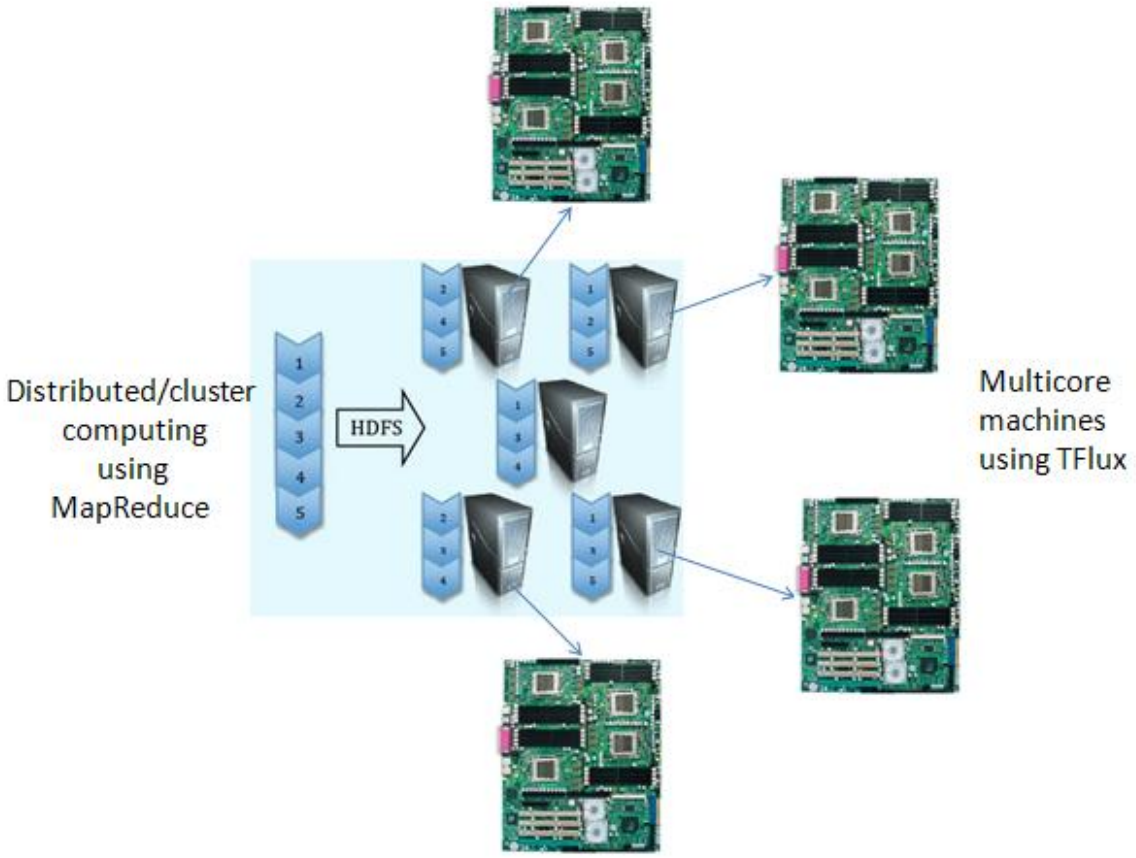
Κεφάλαιο 8 Επίτευξη του στόχου

Στο πλαίσιο της Ατομική Διπλωματικής Εργασίας υλοποιήθηκε παραλληλισμός MapReduce υπολογισμών με την χρήση της πλατφόρμας TFlux χρησιμοποιώντας TFux Directives με παρόμοια μεθοδολογία όπως και στις σχετικές εργασίες που έχουμε αναφέρει. Από τα σενάρια των δυο εφαρμογών που έχουμε τρέξει έχει δηχθεί μια τέτοια υλοποίηση παραλληλισμού Map και Reduce διαδικασιών με TFlux Directives μπορεί να τρέξει αποδοτικά πάνω σε ένα σύστημα με πολλούς επεξεργαστές και να έχει την ίδια απόδοση με την αντίστοιχη υλοποίηση του MapReduce σε συστήματα κοινόχρηστης μνήμης ,του Phoenix.

8.1 Μελλοντική Εργασία

Πέραν του ότι δείχθηκε ότι μπορεί η υλοποίηση παραλληλισμού MapReduce υπολογισμών μέσο της πλατφόρμας TFlux να είναι εφικτή και συνάμα αποδοτική , αφετέρου μπορεί να χχρησιμοποιηθεί ως έναυσμα περισσότερης ερευνητικής μελέτης με σκοπό την βελτίωση της απόδοσης παράλληλης επεξεργασίας ογκώδη δεδομένων με την δημιουργία clusters από multicores ,και στην ουσία της συνεργασίας των δυο εφαρμογών.

Μια πιθανή μελλοντική εργασία επικεντρώνεται κατά κύριο λόγο στην υλοποίηση και την διεξαγωγή πειραμάτων σε clusters με multicore μηχανές όπου πιθανόν σε κάθε σενάριο για κάθε μηχανή θα εφαρμόζεται μια αντίστοιχη εφαρμογή TFlux με αυτή που θα εφαρμόζεται στο σύνολο των μηχανών μιας εφαρμογής MapReduce και συνάμα την σύγκριση της επίδοσης τους με τα cluster που χρησιμοποιούνται σήμερα. Παραστατικά δηλαδή θα έχουμε ένα Distributed / cluster computing στο οποίο θα διεξάγεται ένας MapReduce υπολογισμός συνδυαζόμενος από TFlux υπολογισμούς της εφαρμογής σε κάθε πολυεπεξεργαστικο κόμβο του δικτύου .



Σχήμα 8.1 Συνδυασμός των δυο μοντέλων παραλληλισμού[8,11]

Βιβλιογραφία

[1] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” In: Proc. Of OSDI, 2004.

[2] H. Yang, A. Dasdan, R. Hsiao, Jr., D.S. Parker, “Map-reducemerge: simplified relational data processing on large clusters,” SIGMOD, 2007.

[3] Apache Projects Proceedings : http://hadoop.apache.org/core/

[4] Costas Kyriacou, Paraskevas Evripidou, Pedro Trancoso “Data-Driven Multithreading Using Conventional Multiprocessors”, October 2006

[5] Kyriakos Stavrou, Marios Nikolaides, Demos Pavlou, Samer Arandi, Paraskevas Evripidou, Pedro Trancoso “TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems”, 2008

[6] Kyriakos Stavrou, Demos Pavlou, Marios Nikolaides, Panayiotis Petrides, Paraskevas Evripidou, Pedro Trancoso “Programming Abstractions and Toolchain for Dataflow Multithreading Archtitectures”

[7] “TFlux C Preprocessor (TFluxCpp) -Technical Manual”

[8] Tom White, Book : “Hadoop: The Definitive Guide”

[9] Dean, J. and Ghemawat, S. 2008. “MapReduce: Simplified data processing on large clusters”. San Francisco, CA

[9] “Open Source MapReduce” http://lucene.apache.org/hadoop

[10] Lammal, Ralf. “Google's MapReduce Programming Model Revisited” . http://www.cs.vu.nl/~ralf/MapReduce/paper.pdf

[11] P.Ravindra, V.Deshpande and K.Anyanwu, “Towards Scalable RDF Graph Analytics on MapReduce,”ACM, 2010.

[12] M. Ben-Ari, “Principles of Concurrent and Distributed Programming “

[13] Brian Everitt, Sabine Landau, Morven Leese, “Cluster Analysis”

[14] Programming the TFlux DataDriven Multithreading Processing Platform http://www.hipeac.net/system/files/tflux.pdf

[15] “Introduction to Parallel Programming and MapReduce” http://code.google.com/edu/parallel/mapreduce-tutorial.html

[16] “ Εισαγωγή στη Παράλληλη Επεξεργασία” http://pdplab.it.uom.gr/teaching/llnl-gr/Introduction%20to%20Parallel%20Computing.htm

[17] “Hadoop,Joydeep Sen Sarma”
http://www.facebook.com/note.php?note_id=16121578919

[18] “Yahoo Open-Sources Real-Time MapReduce”
http://gigaom.com/cloud/is-yahoo-set-to-open-source-real-time-mapreduce/

[19] “Module 4: MapReduce”
http://developer.yahoo.com/hadoop/tutorial/module4.html#listmapping

[20] “Threads: Basic Theory and Libraries”
http://www.montegodata.co.uk/CodeUnix/ThreadsBasicTheoryLibraries.htm

[21] “Handling Multiple Processors in Your Code”
Error! Hyperlink reference not valid.

ΠΑΡΑΡΤΗΜΑ Α

wordCount Phoenix source code

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <ctype.h>

#ifdef TBB
#include "tbb/scalable_allocator.h"
#endif

#include "map_reduce.h"
#define DEFAULT_DISP_NUM 10

// a passage from the text. The input data to the Map-Reduce
struct wc_string {
    char* data;
    uint64_t len;
};

// a single null-terminated word
struct wc_word {
    char* data;

    // necessary functions to use this as a key
    bool operator<(wc_word const& other) const {
        return strcmp(data, other.data) < 0;
    }
    bool operator==(wc_word const& other) const {
        return strcmp(data, other.data) == 0;
    }
};


// a hash for the word
struct wc_word_hash
{
    // FNV-1a hash for 64 bits
    size_t operator()(wc_word const& key) const
    {
        char* h = key.data;
        uint64_t v = 14695981039346656037ULL;
        while (*h != 0)
            v = (v ^ (size_t)(*(h++))) * 1099511628211ULL;
        return v;
    }
};

#ifdef MUST_USE_FIXED_HASH
class WordsMR : public MapReduceSort<WordsMR, wc_string, wc_word, uint64_t,
fixed_hash_container<wc_word, uint64_t, sum_combiner, 32768, wc_word_hash
#else
class WordsMR : public MapReduceSort<WordsMR, wc_string, wc_word, uint64_t,
hash_container<wc_word, uint64_t, sum_combiner, wc_word_hash
#endif
#ifdef TBB
    , tbb::scalable_allocator
#endif
```

```
> >
{
    char* data;
    uint64_t data_size;
    uint64_t chunk_size;
    uint64_t splitter_pos;
public:
    explicit WordsMR(char* _data, uint64_t length, uint64_t _chunk_size) :
        data(_data), data_size(length), chunk_size(_chunk_size),
            splitter_pos(0) {}

    void* locate(data_type* str, uint64_t len) const
    {
        return str->data;
    }

    void map(data_type const& s, map_container& out) const
    {
        for (uint64_t i = 0; i < s.len; i++)
        {
            s.data[i] = toupper(s.data[i]);
        }

        uint64_t i = 0;
        while(i < s.len)
        {
            while(i < s.len && (s.data[i] < 'A' || s.data[i] > 'Z'))
                i++;
            uint64_t start = i;
            while(i < s.len && ((s.data[i] >= 'A' && s.data[i] <= 'Z') ||
s.data[i] == '\''))
                i++;
            if(i > start)
            {
                s.data[i] = 0;
                wc_word word = { s.data+start };
                emit_intermediate(out, word, 1);
            }
        }
    }

    /** wordcount split()
     *  Memory map the file and divide file on a word border i.e. a space.
     */
    int split(wc_string& out)
    {
        /* End of data reached, return FALSE. */
        if ((uint64_t)splitter_pos >= data_size)
        {
            return 0;
        }

        /* Determine the nominal end point. */
        uint64_t end = std::min(splitter_pos + chunk_size, data_size);

        /* Move end point to next word break */
        while(end < data_size &&
            data[end] != ' ' && data[end] != '\t' &&
            data[end] != '\r' && data[end] != '\n')
            end++;
```

```
        /* Set the start of the next data. */
        out.data = data + splitter_pos;
        out.len = end - splitter_pos;

        splitter_pos = end;

        /* Return true since the out data is valid. */
        return 1;
    }

    bool sort(keyval const& a, keyval const& b) const
    {
        return a.val < b.val || (a.val == b.val && strcmp(a.key.data,
b.key.data) > 0);
    }
};

#define NO_MMAP

int main(int argc, char *argv[])
{
    int fd;
    char * fdata;
    unsigned int disp_num;
    struct stat finfo;
    char * fname, * disp_num_str;
    struct timespec begin, end;

    get_time (begin);

    // Make sure a filename is specified
    if (argv[1] == NULL)
    {
        printf("USAGE: %s <filename> [Top # of results to display]\n",
argv[0]);
        exit(1);
    }

    fname = argv[1];
    disp_num_str = argv[2];

    printf("Wordcount: Running...\n");

    // Read in the file
    CHECK_ERROR((fd = open(fname, O_RDONLY)) < 0);
    // Get the file info (for file length)
    CHECK_ERROR(fstat(fd, &finfo) < 0);
#ifndef NO_MMAP
#ifdef MMAP_POPULATE
    // Memory map the file
    CHECK_ERROR((fdata = (char*)mmap(0, finfo.st_size + 1,
        PROT_READ, MAP_PRIVATE | MAP_POPULATE, fd, 0)) == NULL);
#else
    // Memory map the file
    CHECK_ERROR((fdata = (char*)mmap(0, finfo.st_size + 1,
        PROT_READ, MAP_PRIVATE, fd, 0)) == NULL);
#endif
#else
    uint64_t r = 0;

    fdata = (char *)malloc (finfo.st_size);
```

```
    CHECK_ERROR (fdata == NULL);
    while(r < (uint64_t)finfo.st_size)
        r += pread (fd, fdata + r, finfo.st_size, r);
    CHECK_ERROR (r != (uint64_t)finfo.st_size);
#endif

    // Get the number of results to display
    CHECK_ERROR((disp_num = (disp_num_str == NULL) ?
      DEFAULT_DISP_NUM : atoi(disp_num_str)) <= 0);

    get_time (end);

#ifdef TIMING
    print_time("initialize", begin, end);
#endif

    printf("Wordcount: Calling MapReduce Scheduler Wordcount\n");
    get_time (begin);
    std::vector<WordsMR::keyval> result;
    WordsMR mapReduce(fdata, finfo.st_size, 1024*1024);
    CHECK_ERROR( mapReduce.run(result) < 0);
    get_time (end);

#ifdef TIMING
    print_time("library", begin, end);
#endif
    printf("Wordcount: MapReduce Completed\n");

    get_time (begin);

    unsigned int dn = std::min(disp_num, (unsigned int)result.size());
    printf("\nWordcount: Results (TOP %d of %lu):\n", dn, result.size());
    uint64_t total = 0;
    for (size_t i = 0; i < dn; i++)
    {
        printf("%15s - %lu\n", result[result.size()-1-i].key.data,
result[result.size()-1-i].val);
    }

    for(size_t i = 0; i < result.size(); i++)
    {
        total += result[i].val;
    }

    printf("Total: %lu\n", total);

#ifndef NO_MMAP
    CHECK_ERROR(munmap(fdata, finfo.st_size + 1) < 0);
#else
    free (fdata);
#endif
    CHECK_ERROR(close(fd) < 0);

    get_time (end);

#ifdef TIMING
    print_time("finalize", begin, end);
#endif

    return 0;
}
```

```
// vim: ts=8 sw=4 sts=4 smarttab smartindent
```

Matrix Multiply Phoenix source code

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <assert.h>

#include "map_reduce.h"

struct mm_data_t {
    int row_num;
    int rows;
    int *matrix_A;
    int *matrix_B;
    int matrix_len;
    int* output;
};

class MatrixMulMR : public MapReduce<MatrixMulMR, mm_data_t, int, int>
{
    int *matrix_A, *matrix_B;
    int matrix_size;
    int row;
    int *output;

public:
    explicit MatrixMulMR(int* _mA, int* _mB, int size, int* out) :
        matrix_A(_mA), matrix_B(_mB), matrix_size(size), row(0),
output(out) {}

    void* locate(mm_data_t* d, uint64_t len) const
    {
        return d->matrix_A + d->row_num * d->matrix_len;
    }

    /** matrixmul_map()
     * Multiplies the allocated regions of matrix to compute partial sums
     */
    void map(mm_data_t const& data, map_container& out) const
    {
        int* a_ptr = data.matrix_A + data.row_num*data.matrix_len + 0;
        int* b_ptr = data.matrix_B + 0;

        int* output = data.output + data.row_num*data.matrix_len;
        for(int i = 0; i < data.matrix_len ; i++) {
            for(int j=0;j<data.matrix_len ; j++) {
                output[j] += a_ptr[i] * b_ptr[j];
            }
            b_ptr += data.matrix_len;
        }
    }

    /** matrixmul_split()
     *  Assign a set of rows of the output matrix to each map task
     * (for now, default to one row per task)
     */
    int split(mm_data_t& out)
    {
```

```
        /* End of data reached, return FALSE. */
        if (row >= matrix_size)
        {
            return 0;
        }

        out.matrix_A = matrix_A;
        out.matrix_B = matrix_B;
        out.matrix_len = matrix_size;
        out.output = output;
        out.rows = 1;
        out.row_num = row++;

        /* Return true since the out data is valid. */
        return 1;
    }
};

int main(int argc, char *argv[]) {

    int i,j, create_files;
    int fd_A, fd_B, file_size;
    int * fdata_A, *fdata_B;
    int matrix_len, row_block_len;
    struct stat finfo_A, finfo_B;
    char const* fname_A, *fname_B;
    int ret;

    struct timespec begin, end;

    get_time (begin);

    srand( (unsigned)time( NULL ) );

    // Make sure a filename is specified
    if (argv[1] == NULL)
    {
        dprintf("USAGE: %s [side of matrix] [size of Row block]\n",
argv[0]);
        exit(1);
    }

    fname_A = "matrix_file_A.txt";
    fname_B = "matrix_file_B.txt";
    CHECK_ERROR ( (matrix_len = atoi(argv[1])) < 0);
    file_size = ((matrix_len*matrix_len))*sizeof(int);

    fprintf(stderr, "***** file size is %d\n", file_size);

    if(argv[2] == NULL)
        row_block_len = 1;
    else
        CHECK_ERROR ( (row_block_len = atoi(argv[2])) < 0);

    if(argv[3] != NULL)
        create_files = 1;
    else
        create_files = 0;

    printf("MatrixMult: Side of the matrix is %d\n", matrix_len);
    printf("MatrixMult: Row Block Len is %d\n", row_block_len);
```

```
    printf("MatrixMult: Running...\n");

    /* If the matrix files do not exist, create them */
    if(create_files)
    {
        dprintf("Creating files\n");

        int value = 0;
        CHECK_ERROR((fd_A = open(fname_A,O_CREAT | O_RDWR,S_IRWXU)) < 0);
        CHECK_ERROR((fd_B = open(fname_B,O_CREAT | O_RDWR,S_IRWXU)) < 0);

        for(i=0;i<matrix_len;i++)
        {
            for(j=0;j<matrix_len;j++)
            {
                value = (rand())%11;
                ret = write(fd_A,&value,sizeof(int));
                assert(ret == sizeof(int));
                //dprintf("%d  ",value);
            }
            //dprintf("\n");
        }
        //dprintf("\n");

        for(i=0;i<matrix_len;i++)
        {
            for(j=0;j<matrix_len;j++)
            {
                value = (rand())%11;
                ret = write(fd_B,&value,sizeof(int));
                assert(ret == sizeof(int));
                //dprintf("%d  ",value);
            }
            //dprintf("\n");
        }

        CHECK_ERROR(close(fd_A) < 0);
        CHECK_ERROR(close(fd_B) < 0);
    }

    // Read in the file
    CHECK_ERROR((fd_A = open(fname_A,O_RDONLY)) < 0);
    // Get the file info (for file length)
    CHECK_ERROR(fstat(fd_A, &finfo_A) < 0);
#ifndef NO_MMAP
#ifdef MMAP_POPULATE
    // Memory map the file
    CHECK_ERROR((fdata_A = (int*)mmap(0, finfo_A.st_size + 1,
        PROT_READ, MAP_PRIVATE | MAP_POPULATE, fd_A, 0)) == NULL);
#else
    // Memory map the file
    CHECK_ERROR((fdata_A = (int*)mmap(0, finfo_A.st_size + 1,
        PROT_READ, MAP_PRIVATE, fd_A, 0)) == NULL);
#endif
#else
    int ret;

    fdata_A = (char *)malloc (file_size);
    CHECK_ERROR (fdata_A == NULL);

    ret = read (fd_A, fdata_A, file_size);
```

```
    CHECK_ERROR (ret != file_size);
#endif

    // Read in the file
    CHECK_ERROR((fd_B = open(fname_B,O_RDONLY)) < 0);
    // Get the file info (for file length)
    CHECK_ERROR(fstat(fd_B, &finfo_B) < 0);
#ifndef NO_MMAP
#ifdef MMAP_POPULATE
    // Memory map the file
    CHECK_ERROR((fdata_B = (int*)mmap(0, finfo_B.st_size + 1,
        PROT_READ, MAP_PRIVATE | MAP_POPULATE, fd_B, 0)) == NULL);
#else
    // Memory map the file
    CHECK_ERROR((fdata_B = (int*)mmap(0, finfo_B.st_size + 1,
        PROT_READ, MAP_PRIVATE, fd_B, 0)) == NULL);
#endif
#else
    fdata_B = (char *)malloc (file_size);
    CHECK_ERROR (fdata_B == NULL);

    ret = read (fd_B, fdata_B, file_size);
    CHECK_ERROR (ret != file_size);
#endif

    int* output = (int*)calloc(matrix_len*matrix_len, sizeof(int));

    fprintf(stderr, "***** data size is %ld\n", (intptr_t)file_size);
    printf("MatrixMult: Calling MapReduce Scheduler Matrix
Multiplication\n");

    get_time (end);
    print_time("initialize", begin, end);

    get_time (begin);
    std::vector<MatrixMulMR::keyval> result;
    MatrixMulMR mapReduce(fdata_A, fdata_B, matrix_len, output);
    mapReduce.run(result);
    get_time (end);
    print_time("library", begin, end);

    get_time (begin);
    int sum = 0;
    for(i=0;i<matrix_len*matrix_len;i++)
    {
          sum += output[i];
    }
    printf ("MatrixMult: total sum is %d\n", sum);

    printf("MatrixMult: MapReduce Completed\n");

    free(output);

#ifndef NO_MMAP
    CHECK_ERROR(munmap(fdata_A, file_size + 1) < 0);
#else
    free (fdata_A);
#endif
    CHECK_ERROR(close(fd_A) < 0);

#ifndef NO_MMAP
```

```
    CHECK_ERROR(munmap(fdata_B, file_size + 1) < 0);
#else
    free (fdata_B);
#endif
    CHECK_ERROR(close(fd_B) < 0);

    get_time (end);
    print_time("finalize", begin, end);

    return 0;
}

// vim: ts=8 sw=4 sts=4 smarttab smartindent
```

ΠΑΡΑΡΤΗΜΑ Β

Install and Run TFlux

3. Execution components

This section describes the installation process of TFluxCpp together with usage instructions.

3.1 Execution Environment

TFluxCpp runs on both UNIX-based systems and Windows-based systems (using Cygwin).

3.2 Building TFluxCpp

To install TFluxCpp execute the following commands:

```
$> tar -xvf tfluxcpp.tar.gz
$> cd tfluxcpp
$> make clean
$> make tfluxcpp
```

3.3 Preprocessing programs

TFluxCpp is a command line tool and all the operations it offers can be invoked through its command line interface. The command line interface of TFluxCpp is as shown below. The several options (summarized in Table 1) are analyzed in the following paragraphs.

```
$>tfluxcpp [-t targetSystem] [-o <outputFile>] [-i <inputFile>] [-c] [-p] [-fs T C I] [-s]
```

[-t] Target Architecture

This option defines the architecture for which TFluxCpp produces the output C code. Currently TFluxCpp supports two architectures, TFluxHard and TFluxSoft. Note however, that TFluxHard exists only at simulation level.

-t hard To produce code for TFluxHard. Note that this system exists only at simulation level so this option is not to be used.

-t soft To produce code of TFluxSoft. This implementation runs on most Linux-based Operating Systems and can be downloaded from www.cs.ucy.ac.cy/carch/casper/tflux

[-o] Output File

Defines the file where TFluxCpp will save the produced C program.

[-i] Input File

Defines the input file that contains the TFlux program (C code augmented with the TFl

[-c] Compress DThread Identifier numbers

This option enables an internal operation of TFluxCpp for "compressing" the Thread automatically produced DThreads. This option should *always be used*.

[-p] Print the Synchronization Graph

This option is used to print the synchronization graph of the TFlux application given as input to TFluxCpp. This graph represents the dependencies between the DThreads. In addition, it provides information about the Kernel (the Kernel is the parallel entity executing DThreads - details about Kernels are presented in Section 4.2) that executes each DThread and the block (see Section 4.3) each DThread belongs to.

The example that follows regards a part of the synchronization graph of a sample program. In this example DThread 10001 belongs to block 1.

```
================================
[  B L O C K    1               ]
================================

[I] TID:10001 BLC:1 KID:1 RC:3
        Consumer list [TID] : 1/0/0  #  10002/0/0  #  10003/0/0  #  10004/0/0
  Threads to load list [TID] : 1/0/0  #  2/0/0  #  25001/0/0  #  20001/0/0

.........more threads are printed below.........
```

The first identifier (*[I]*) shows the type of the DThread. The possible types are:

I	**Inlet DThread**	DThread that loads the metadata of DThreads into the scheduler. The Inlet DThreads are automatically created by TFluxCpp.
O	**Outlet DThread**	DThread that clears the allocated resources (for the DThreads' metadata) from the scheduler. Outlet DThreads are automatically created by TFluxCpp.
E	**Execution DThread**	DThread that executes the application's code
L	**Loop DThread**	A special type of execution DThreads for the execution of parallel loops
R	**Reduction DThread**	DThread for executing the reduction operation of loops. Reduction DThreads are automatically created by TFluxCpp when necessary.

What follows is the DThread's description. For each DThread the output includes its Thread ID *(in the above example TID = 1)*, the block in which it belongs *(in the above example BLC = 1)*, the Kernel that executes it (*in the above example* KID = 1) and the ready count (*in the above example* RC = 3) which is the number DThreads that the specific DThread depends on.

TID	DThread Identifier
BLC	The block identifier in which the DThread belongs
KID	The identifier of the Kernel that executes the DThread
RC	The ready count value of a DThread, i.e. the number of DThread this particular DThread depends on.

The second line corresponds to the *consumer list* which is a list with the DThreads which need to wait for the execution of this particular DThread to complete prior to initiating their execution.

The third line is for the *DThreads to load list*, a list that indicates the DThreads this particular DThread must load into the scheduler. Note that it is only a special category of DThreads, the *inlet DThreads* that load the data. Each DThread in these lists is expressed in the form of Thid/Cntx/Iter where Thid is the identifier of the DThread, Cntx is the context Id and Iter the iteration Id (for the common user, the identifier that is important is the THID). For example 1/0/0 indicates a DThread where Thid is 1, Cntx is 0 and Iter is 0.

The *Thread Id* is used in order to define uniquely each thread. It is defined by the user during the development of the TFlux application. It is also used in order to define the dependencies between threads. The *Context Id* will be used in the future in order to execute nested loops. Finally the *Iteration Id* expresses the iteration of the loop that particular instance of the DThread executes.

[-fs N N N] Field Sizes

This option allows the user to define the sizes of the Thread ID, Context ID, Iteration ID and fields. If the programmer does not define these values they will be automatically created by the preprocessor. It is recommended to allow TFluxCpp to automatically define these values.

However, if the programmer decides to explicitly define these values they should be defined as follows. The sum of these three values should always be equal to 32. The Iteration ID field size (I) should be such that $2^I \geq X$ where X is the maximum number of iterations for the all the loops that have been parallelized with the *ddm for* directive (explained in Section 5.5). For example, if such a program includes a parallel loop that executes 10^6 iterations the Iteration ID field size should be equal to 20 ($2^{20}>10^6$ and $2^{19}<10^6$). The remaining 12 bits should be assigned to the Context ID and Thread ID fields. However, given that for the current version of TFluxCpp Context ID can be as small as 1 bit, the programmer should use the remaining 11 bits for the Thread ID field.

[-s] Print Summary
Prints a summary of the preprocessing phase.

Table 1: TFlux Preprocessor Command Line Interface Summary

Options	Usage	Optional
-t	Specify the architecture for which the produced program is targeted to. The supported architectures are: • **hard** : targets the hardware implementation of TFlux • **soft** : targets the software implementation of TFlux	No
-o	Defines the file in which the produced C program will be saved.	No
-i	Defines the input program (C code with TFlux Directives)	No
-c	Compacts the field sizes (always use this option, not using it is for debugging purposes only)	Yes
-p	Prints the synchronization graph of the program (dependencies between the DThreads) and other information regarding the DThreads.	Yes
-fs	Definition of sizes of the Thread ID, Context ID and Iteration ID fields in bits. The sum of these three values should be equal to 32. If this option is not used, the preprocessor will automatically calculate the necessary field sizes • *ThreadID ContextID IterationID*	Yes
-s	Prints preprocessing summary	Yes

Usage Examples
A set of typical usage examples is listed below.

```
./tfluxcpp -o outputProgram.c -i inputProgram.ddm -c
./tfluxcpp -o outputProgram.c -i inputProgram.ddm -c -fs 7 1 24
./tfluxcpp -o outputProgram.c -i inputProgram.ddm -c -p
./tfluxcpp -o outputProgram.c -i inputProgram.ddm -c -s
./tfluxcpp -o outputProgram.c -i inputProgram.ddm -c -p -s
./tfluxcpp -o outputProgram.c -i inputProgram.ddm -c -p -fs 7 1 24
```