Ατομική Διπλωματική Εργασία

Evaluating Hadoop, a MapReduce implementation, for Multicore systems.

Κωνσταντίνος Χριστοφή

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2009

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Evaluating Hadoop, a MapReduce implementation,
for multicore systems

Κωνσταντίνος Χριστοφή

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 2009

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Pedro Trancoso για την υποστήριξη και καθοδήγηση που μου προσέφερε κατά την διάρκεια της υλοποίησης της παρούσας εργασίας καθώς και για την επίτευξη μιας άψογης συνεργασίας.

Επίσης ευχαριστίες πρέπει να δοθούν στον διδακτορικό φοιτητή του τμήματος, Παναγιώτη Πετρίδη, ο οποίος βρισκόταν εκεί καθ' όλη της διάρκεια της χρονιάς για να με συμβουλεύει μέσα από εποικοδομητικές συζητήσεις για τα όποια αμφιλεγόμενα ή όχι σημεία τη διπλωματικής μου εργασίας.

Ακόμη, θα ήθελα να ευχαριστήσω τους συνεργάτες - συμφοιτητές μου από το CASPER Group του επιβλέποντα καθηγητή μας, οι οποίοι φρόντιζαν να υπάρχει πάντα ένα ευχάριστο περιβάλλον εργασίας στο εργαστήριο.

Τέλος θα ήθελα να ευχαριστήσω την οικογένεια και τους φίλους μου για τη στήριξη που μου έδιναν καθ' όλη τη διάρκεια αυτής της χρονιάς.

Περίληψη

Το προγραμματιστικό μοντέλο MapReduce προτάθηκε από την Google το 2003. Σκοπός του είναι να επεξεργάζεται μεγάλα δεδομένα εισόδου και να παράγει τα δεδομένα εξόδου χρησιμοποιώντας ένα σύνολο από διασυνδεδεμένους υπολογιστές (cluster) παραλληλίζοντας αυτόματα την εκτέλεση της εργασίας πάνω σε αυτό το δίκτυο υπολογιστών.

Στο πλαίσιο αυτής της Ατομική Διπλωματικής Εργασίας θα ασχοληθώ με μια συγκεκριμένη υλοποίηση του προγραμματιστικού μοντέλου MapReduce, το Hadoop μιας υλοποίησης ανοιχτού κώδικα γραμμένη σε Java, και θα την μετατρέψω έτσι που να μπορεί να τρέξει αποδοτικά πάνω σε ένα σύστημα με πολλούς επεξεργαστές (multicore) και όχι πάνω σε ένα δίκτυο από διασυνδεδεμένους υπολογιστές (cluster). Ακόμα θα κάνω μια ανάλυση της εκτέλεσης των εφαρμογών σε σύστημα με πολλούς επεξεργαστές για να μελετήσω αν είναι δυνατή η αντικατάσταση των συστημάτων τύπου cluster με συστήματα πολυεπεξεργαστών.

Καταλήγω στο συμπέρασμα ότι ο Hadoop είναι μια νέα τάση στην παράλληλη επεξεργασία και φαίνεται να είναι αρκετά υποσχόμενη για χρήση πάνω σε συστήματα πολυεπεξεργαστών για τις εφαρμογές που δοκιμάστηκαν. Ακόμα καταλήγω ότι τα συστήματα πολυεπεξεργαστών είναι αρκετά αποδοτικά για χρήση με το Hadoop και ίσως μια καλή σκέψη να είναι να τρέχει στο μέλλον να έχουμε clusters αποτελούμενα από multicore μηχανές έτσι που να μπορούμε να συνδυάσουμε σε ένα σύστημα τα θετικά στοιχεία και των δύο εγκαταστάσεων.

Περιεχόμενα

Κεφάλαιο 1 Εισαγωγή 1
1.1 Κίνητρο 1
1.2 Το προγραμματιστικό μοντέλο MapReduce 2
1.3 Επισκόπηση Hadoop 4
1.4 Το Hadoop πάνω σε σύστημα πολυεπεξεργαστών 6
1.5 Οργάνωση 7
Κεφάλαιο 2 Σχετική Εργασία 8
2.1 Phoenix MapReduce 8
2.2 MapReduce Cell 9
2.3 Mars 10
Κεφάλαιο 3 Hadoop 12
3.1 Εντολές χρήσης Hadoop 12
3.2 Πώς να γράψετε ένα πρόγραμμα Hadoop 14
3.3 Πώς να τρέξετε ένα πρόγραμμα Hadoop 17
Κεφάλαιο 4 Αρχιτεκτονικές και πλατφόρμες 18
4.1 Αρχιτεκτονική Hadoop cluster 18
4.2 Αρχιτεκτονική HDFS 21
4.3 Αρχιτεκτονική multicore εγκατάστασης 22
4.4 Multicore επεξεργαστές και παράλληλος προγραμματισμός 23
Κεφάλαιο 5 Αποτελέσματα πειραμάτων 24
5.1 Περιβάλλον εγκατάστασης 24
5.2 Τρόπος συλλογής αποτελεσμάτων 24
5.3 Δοκιμαστικές εφαρμογές 26
5.4 Αποτελέσματα multicore εγκατάστασης 29
5.4.1 WordCount 29
5.4.2 TeraGen 32
5.4.3 TeraSort 35
5.5 Αποτελέσματα cluster εγκατάστασης 37
5.6 Αποτελέσματα εκτέλεσης σε ψευδό-δίκτυο (pseudo-cluster) 39
5.7 Σύγκριση αποτελεσμάτων 40

Κεφάλαιο 6 Συμπεράσματα και μελλοντική εργασία 42
6.1 Συμπεράσματα 42
6.2 Μελλοντική Εργασία 44
Παράρτημα Α 46
Παράρτημα Β 49

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο 1
1.2 Το προγραμματιστικό μοντέλο MapReduce 2
1.3 Επισκόπηση Hadoop 4
1.4 Το Hadoop πάνω σε σύστημα πολυεπεξεργαστών 6
1.5 Οργάνωση 7

1.1 Κίνητρο

Η σημερινή τάση στην κατασκευή νέων επεξεργαστών είναι η αύξηση του αριθμού των επεξεργαστών που τοποθετούνται πάνω σε ένα chip παρά η αύξηση της συχνότητας ενός επεξεργαστή. Αυτό οφείλεται κυρίως στο πρόβλημα της κατανάλωσης ενέργειας από τον επεξεργαστή, αλλά και στην υπερθέρμανση του επεξεργαστή από τυχόν αυξημένη συχνότητα αλλά και περιπλοκότητα του σχεδίου. Η εκμετάλλευση αυτής της υπολογιστικής ισχύς μέσω throughput (δηλαδή η παράλληλη εκτέλεση αριθμού προγραμμάτων ή processes) είναι εύκολη, αλλά το πιο σημαντικό, και πιο δύσκολο, είναι η εκμετάλλευσή της για την εκτέλεση ενός μόνο προγράμματος, παραλληλοποιώντας το. Προς αυτή την κατεύθυνση έχουν γίνει αρκετά θετικά βήματα αλλά τα περισσότερα αφορούν τον τομέα της παράλληλης επεξεργασίας σε cluster από υπολογιστές αντί σε multicore μηχανές. Μια τέτοια προσέγγιση είναι το προγραμματιστικό μοντέλο MapReduce το οποίο προτάθηκε από την Google για την παράλληλη επεξεργασία τεράστιων συνόλων δεδομένων εισόδου προσφέροντας στον χρήστη αυτόματο παραλληλισμό της εφαρμογής του αναλαμβάνοντας τις λεπτομέρειες του παραλληλισμού. Στα πλαίσια αυτής της ατομικής διπλωματικής εργασίας θα ασχοληθώ με το Hadoop μιας ανοιχτού κώδικα υλοποίησης του μοντέλου MapReduce

από την Yahoo! μετατρέποντας το έτσι που να μπορεί να τρέξει αποδοτικά πάνω σε μια multicore μηχανή.

1.2 Το προγραμματιστικό μοντέλο MapReduce

Το MapReduce είναι ένα μοντέλο υπολογισμού για επεξεργασία και παραγωγή τεραστίων συνόλων δεδομένων που προτάθηκε για πρώτη φορά από το Google το 2004 στο άρθρο των Jeffrey Dean και Sanjay Ghemawat[4]. Αρχικός σκοπός του μοντέλου MapReduce ήταν η επεξεργασία τεραστίων συνόλων δεδομένων πάνω σε ένα σύνολο από δικτυωμένους οικιακούς υπολογιστές (cluster of commodity machines). Για την επίλυση κάποιου προβλήματος με το MapReduce, ο προγραμματιστής πρέπει να υλοποιήσει τουλάχιστο την συνάρτηση ***map***, η οποία επεξεργάζεται ένα ζεύγος κλειδιού/τιμής για να παράξει ένα ενδιάμεσο ζεύγος κλειδιού/τιμής, και την συνάρτηση ***reduce***, η οποία ενώνει όλες τις ενδιάμεσες τιμές που σχετίζονται με το ίδιο ενδιάμεσο κλειδί.

Το προγραμματιστικό μοντέλο, που χρησιμοποιείται για την υλοποίηση του MapReduce, είναι αρκετά απλό και γενικό. Δηλαδή για οποιοδήποτε πρόβλημα θα λυθεί με το μοντέλο η ιδέα της υλοποίησης και ο αλγόριθμος που ακολουθείται είναι κοινός. Η ιδέα της λειτουργίας του MapReduce είναι να παίρνει σαν είσοδο ένα σύνολο από ζευγάρια κλειδιών εισόδου και στην έξοδο να παράγει ένα σύνολο από ζευγάρια κλειδιών εξόδου. Ο χρήστης του MapReduce εκφράζει την πράξη αυτή σαν δύο συναρτήσεις/μεθόδους, την ***map*** και την ***reduce***. Ο προγραμματιστής το μόνο που έχει να κάνει είναι να υλοποιήσει τις δύο αυτές συναρτήσεις.

Η ***map*** συνάρτηση παίρνει σαν είσοδο ένα ζεύγος κλειδιού και, αφού εφαρμόσει την συνάρτηση του χρήστη πάνω τους, παράγει ένα ενδιάμεσο ζεύγος κλειδιού, το οποίο στην συνέχεια θα δώσει στην συνάρτηση ***reduce***, η οποία με τη σειρά της ενώνει όλες αυτές τις ενδιάμεσες τιμές που σχετίζονται με το ίδιο ενδιάμεσο κλειδί.

Η συνάρτηση ***reduce*** προσπαθεί να συγχωνέψει όλες αυτές τις τιμές εφαρμόζοντας τον κώδικα του χρήστη, για να μπορέσει πιθανώς να δημιουργήσει ένα μικρότερο σύνολο τιμών. Συνήθως δημιουργείται μια τιμή εξόδου για κάθε εκτέλεση της ***reduce***.

Παρακάτω παρατίθεται ένα μικρό και απλό παράδειγμα υλοποίησης των συναρτήσεων ***map*** και ***reduce*** σε ψευδοκώδικα. Το παρακάτω παράδειγμα παίρνει σαν είσοδο διάφορα αρχεία κειμένου και μετρά τις εμφανίσεις κάθε λέξης μέσα σε όλα τα αρχεία που έχει πάρει σαν είσοδο. Η ***map,*** αφού πάρει τα αρχεία, δίνει σε κάθε εμφάνιση μιας λέξης μια προσωρινή τιμή ίση με ένα. Όταν η ***map*** τελειώσει την εκτέλεση της και δώσει τα αποτελέσματα της στην ***reduce,*** τότε η ***reduce*** θα αθροίσει όλες αυτές τις προσωρινές μεταβλητές για κάθε μοναδική λέξη και θα μας δώσει πίσω σαν αποτέλεσμα τον αριθμό εμφάνισης της κάθε μοναδικής λέξης μέσα στα αρχεία.

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

**ο πιο πάνω ψευδοκώδικας είναι από το άρθρο «Evaluating MapReduce for Multi-core and Multiprocessor Systems»*[1]

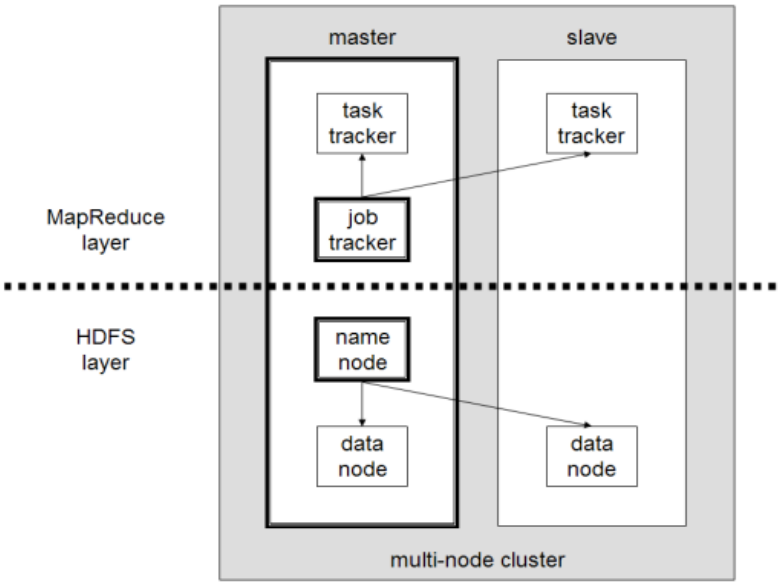
Αυτό που έχει να κερδίσει ένας προγραμματιστής μέσα από την χρήση του μοντέλου αυτού είναι η απλότητα του κώδικα και του αλγορίθμου. Ο ίδιος δε χρειάζεται να ασχοληθεί καθόλου με τον παραλληλισμό της εκτέλεσης του αλγόριθμου αυτού. Ό,τι αφορά την παράλληλη εκτέλεση, τη συνοχή των δεδομένων και αποτελεσμάτων, αλλά και την επικοινωνία επαφίονται στο ***runtime system***. Συνεπώς, ο κώδικας γίνεται

εύκολα φορητός σε διαφορετικά συστήματα. Οι κώδικες του MapReduce εύκολα μπορούν να παραλληλιστούν, αφού η ***map*** μπορεί να εκτελεστεί με τα ίδια δεδομένα σε πολλές μηχανές. Η ***reduce,*** αφού έχει πάρει τα αποτελέσματα της ***map,*** μπορεί να δημιουργηθεί σε πολλά αντίγραφα με τα ίδια δεδομένα ή ακόμα να μοιραστούν τα δεδομένα στα διάφορα αντίγραφα και να εκτελεστούν παράλληλα.

1.3 Επισκόπηση Hadoop

Το ***Hadoop***, είναι μια πολύ σημαντική υλοποίηση του μοντέλου MapReduce ανοιχτού κώδικα (open source) από την Yahoo! γραμμένη σε Java, η οποία έχει ως κύριο σκοπό την επίλυση προβλημάτων με μεγάλα δεδομένα εισόδου πάνω σε ένα δίκτυο από διασυνδεδεμένους οικιακούς υπολογιστές (cluster). Για την λειτουργία του Hadoop είναι απαραίτητη η αποθήκευση των δεδομένων εισόδου, αλλά και των αποτελεσμάτων, στο HDFS (Hadoop's Distributed File System), το οποίο είναι ένα κατανεμημένο σύστημα αποθήκευσης εξαιρετικά μεγάλων δεδομένων σε πολλαπλές μηχανές, που ανήκουν σε ένα δίκτυο (cluster) παρόμοιο με το GFS (Google File System).

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. Ο χρήστης στέλνει τις map και reduce εργασίες του στον jobtracker, ο οποίος τις τοποθετεί σε μια σειρά από έτοιμες να εκτελεστούν εργασίες και τις εξυπηρετεί με την πολιτική FIFO (First In First Out) στέλλοντας τις στους tasktrackers για εκτέλεση. Ο κάθε tasktracker εκτελεί απλώς τις εργασίες που του αναθέτει ο jobtracker. Ακόμα υπάρχει ένας κεντρικός «***namenode***», ο οποίος κρατάει την θέση του κάθε δεδομένου μέσα στο HDFS και πολλοί «***datanodes***», πάλι ένας σε κάθε κόμβο του δικτύου μας, οι οποίοι αποθηκεύουν τα δεδομένα.



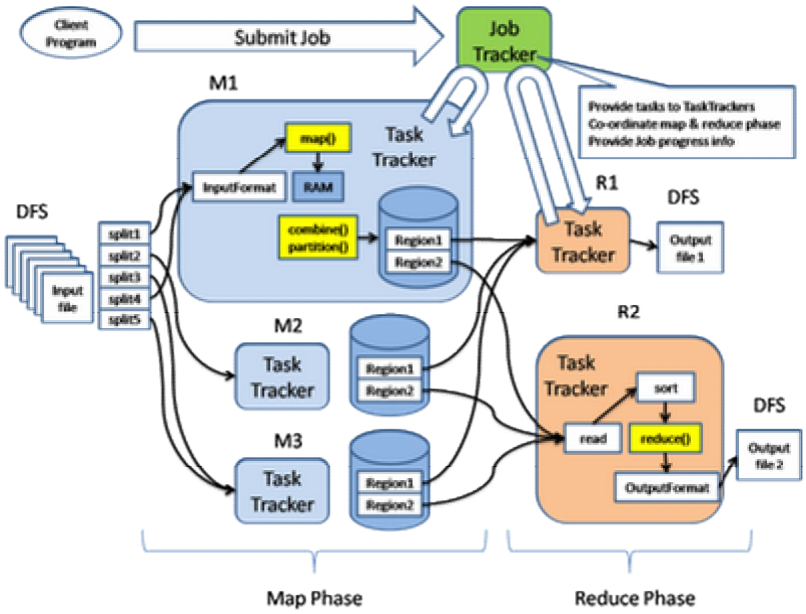
Σχήμα 1.1 Αρχιτεκτονική Hadoop cluster

Στη φάση map της εκτέλεσης το Hadoop «σπάζει» τα δεδομένα εισόδου σε παρά πολλά πιο μικρά κομμάτια, ανάλογα από το αρχικό μέγεθος τους αλλά και από το μέγεθος του δικτύου μας, και αναθέτει το κάθε κομμάτι σε μια map εργασία. Στην συνέχεια διαμοιράζει τις map εργασίες μέσω του jobtracker στους υπολογιστές, που βρίσκονται μέσα στο δίκτυο, για εκτέλεση. Μετά την εκτέλεση της, η κάθε map εργασία γράφει το ενδιάμεσο αποτέλεσμα της στο HDFS στο τοπικό datanode του υπολογιστή, που έγινε η εκτέλεση.

Μετά την εκτέλεση των map εργασιών το Hadoop ταξινομεί τα ενδιάμεσα αποτελέσματα, με βάση το κλειδί τους, έτσι όλες οι τιμές, πού αναφέρονται στο ίδιο κλειδί, να εμφανίζονται μαζί διευκολύνοντας την φάση του reduce, που θα ακολουθήσει. Στην συνέχεια «σπάζει» τα ενδιάμεσα δεδομένα σε τόσα κομμάτια όσα και οι reduce εργασίες, που θα ακολουθήσουν.

Στη φάση reduce, που ακολουθεί, ο jobtracker στέλνει τις reduce εργασίες όσο πιο κοντά στα δεδομένα. Αν είναι δυνατό αναθέτει την εκτέλεση στον κόμβο, που έχει αποθηκευμένα τα ενδιάμεσα δεδομένα. Κάθε reduce εργασία εφαρμόζει την συνάρτηση reduce του χρήστη πάνω στο κομμάτι των ενδιάμεσων δεδομένων, που της έχουν ανατεθεί, και παράγει το τελικό αποτέλεσμα.

Η κάθε φάση εκτελείται με διαχείριση των σφαλμάτων. Αν ένας κόμβος αποτύχει την ώρα πού εκτελεί μια εργασία, τότε το Hadoop αναθέτει την ίδια εργασία σε ένα άλλο κόμβο του δικτύου για επανεκτέλεση.



Σχήμα 1.2 Οι φάσεις εκτέλεσης του Hadoop

Διατηρώντας μεγάλο αριθμό από map και reduce εργασίες το Hadoop επιτυγχάνει καλό load balancing και σε περιπτώσεις οπού υπάρχει σφάλμα η επανεκτέλεση της συγκεκριμένης εργασίας έχει μικρό overhead.

Το Hadoop σήμερα είναι η πιο διαδεδομένη υλοποίηση του MapReduce και χρησιμοποιείται για διδακτικούς σκοπούς σε αρκετά πανεπιστήμια του κόσμου, αλλά και σε μεγάλους οργανισμούς ανά το παγκόσμιο για την επεξεργασία μεγάλων δεδομένων εισόδου. Κάποιοι από τους οργανισμούς, που διατηρούν clusters για εκτέλεση Hadoop εργασιών, είναι: Yahoo!, Amazon, AOL, Alibaba, Cornell University Web Lab, ETH Zurich Systems Group, Facebook, Google, IBM, New York Times κ.α.

1.4 Το Hadoop πάνω σε σύστημα πολυεπεξεργαστών

Για να καταστεί δυνατή η λειτουργία του Hadoop πάνω σε ένα σύστημα με πολυεπεξεργαστές, και για να δουλεύει αποδοτικά διατηρώντας όμως την λειτουργία

του με βάση το μοντέλο MapReduce, έπρεπε όλα τα δομικά στοιχεία του (jobtracker, tasktracker, namenode και datanode) να τρέχουν στην ίδια μηχανή και να μπορούν να επικοινωνούν μεταξύ τους.

Για το σκοπό αυτό προσομοίωσα την ύπαρξη ξεχωριστών κόμβων μέσα σε δίκτυο στέλνοντας όλα τα μηνύματα της μεταξύ τους επικοινωνίας πάνω στην ίδια μηχανή (με χρήση του localhost) υιοθετώντας ξεχωριστό αριθμό θύρας (port number) για κάθε στοιχείο.

Η πιο πάνω προσέγγιση επιτρέπει τη συνύπαρξη ολόκληρου του δικτύου εκτέλεσης του Hadoop πάνω σε μια μηχανή με πολυεπεξεργαστές και την εκτέλεση των map και reduce εργασιών παράλληλα πάνω στους επεξεργαστές της. Έτσι είναι εφικτή η εκμετάλλευση μερικών από τα γνωρίσματα των μηχανών διαμοιραζόμενης μνήμης (shared memory multiprocessor machines), όπως η κοινή κρυφή μνήμη δεύτερου επιπέδου (L2 cache), η κοινή κύρια μνήμη (RAM) και ο ομοιόμορφος χρόνος πρόσβασης των διεργασιών στον σκληρό δίσκο για ανάγνωση ή/και για εγγραφή δεδομένων, για βελτίωση της απόδοσης της εγκατάσταση.

1.5 Οργάνωση

Στο επόμενο κεφάλαιο (Κεφάλαιο 2) θα ασχοληθώ με την σχετική εργασία που έγινε μέχρι σήμερα στον τομέα του προγραμματιστικού μοντέλου MapReduce. Στην συνέχεια στο Κεφάλαιο 3 θα ασχοληθώ με την περιγραφή των αρχιτεκτονικών, που θα χρησιμοποιήσω. Στο Κεφάλαιο 4 θα ασχοληθώ με το Hadoop και την χρήση του. Στο Κεφάλαιο 5 θα παρουσιάσω τα αποτελέσματα των πειραμάτων μου και τέλος στο Κεφάλαιο 6 θα παρουσιάσω τα συμπεράσματα μου.

Κεφάλαιο 2

Σχετική Εργασία

2.1 Phoenix MapReduce 8
2.2 MapReduce Cell 9
2.3 Mars 10

2.1 Phoenix MapReduce

Το Phoenix[1] είναι μια υλοποίηση του προγραμματιστικού μοντέλου MapReduce και βασίζεται στις ίδιες αρχές, αλλά στοχεύει στον παραλληλισμό σε συστήματα διαμοιραζόμενης μνήμης (shared memory systems), όπως multicore systems και symmetric multiprocessors. Στην υλοποίηση περιλαμβάνεται ένα προγραμματιστικό API και ένα σύστημα χρόνου εκτέλεσης (runtime system). Είναι μία αρκετά αποδοτική υλοποίηση και επιτυγχάνει την παραλληλοποίηση του κώδικα με την χρήση Pthreads. Το Phoenix χρησιμοποίει threads για την εκτέλεση πολλαπλών παράλληλων map και reduce εργασιών. Ακόμα χρησιμοποιεί κοινά buffers, ώστε να κάνει την επικοινωνία των διάφορων εργασιών πιο αποδοτική και πιο εύκολη, χωρίς αχρείαστη αντιγραφή δεδομένων. Το σύστημα χρόνου εκτέλεσης (runtime system) ελέγχεται από τον χρονοπρογραμματιστή (scheduler), ο οποίος δημιουργεί και διαχειρίζεται τα threads, τα οποία εκτελούν όλες τις map και reduce εργασίες, και ακόμα διαχειρίζεται τα κοινά buffers για την επικοινωνία των εργασιών. Για κάθε επεξεργαστή ο scheduler δημιουργεί ένα thread στο οποίο αναθέτει δυναμικά map και reduce εργασίες.

Για να αρχίσει η φάση του map, ο scheduler κάνει χρήση μια συνάρτησης, της *splitter,* η οποία μοιράζει τα δεδομένα εισόδου σε ίσα κομμάτια, για να επεξεργαστούν από τις map εργασίες, που ανατίθενται δυναμικά σε κάθε thread και παράγουν τα ενδιάμεσα δεδομένα. Ο scheduler πρέπει να περιμένει όλες τις map εργασίες να ολοκληρωθούν πριν αρχίσει τις reduce εργασίες. Στην συνέχεια κάνει χρήση της συνάρτησης *partition,*

η οποία χωρίζει τα ενδιάμεσα δεδομένα σε κομμάτια διασφαλίζοντας πως τα δεδομένα με το ίδιο κλειδί θα καταλήξουν στο ίδιο κομμάτι και αναθέτει δυναμικά τις reduce εργασίες στα threads. Στην τελευταία φάση τα τελικά δεδομένα εξόδου ενώνονται σε ένα buffer ταξινομημένα ανά κλειδί. Το runtime system αναλαμβάνει την αυτόματη ανάκαμψη από προσωρινά ή και μόνιμα σφάλματα εκτέλεσης μια εργασίας. Μια εργασία, η οποία καθυστερεί ή αποτυγχάνει να ολοκληρωθεί, επαναλαμβάνεται στον ίδιο επεξεργαστή, εάν αυτό είναι δυνατό, ή αν η αποτυχία προέκυψε από υλικό που επαναπρογραμματίζεται σε άλλο επεξεργαστή, για να μειωθούν όσο το δυνατό περισσότερο οι καθυστερήσεις στην εκτέλεση της συνολικής εργασίας.

2.2 MapReduce Cell

Ο Cell είναι μια μηχανή με τεράστιες δυνατότητες, στην οποία η μέθοδος του ***mapReduce*** ταιριάζει απόλυτα. Αλλά όπως πάντα τίποτα δεν είναι τόσο απλό όσο φαίνεται. Ο Cell, παρόλες τις δυνατότητες που διαθέτει, έχει το ‘‘μειονέκτημα’’ ότι όλα πρέπει να γίνουν από τον προγραμματιστή. Δηλαδή, ο συγχρονισμός των SPEs και του PPE, καθώς επίσης και όλες οι μεταφορές δεδομένων από τη μνήμη των SPEs στην κύρια μνήμη, πρέπει να γίνουν από τον προγραμματιστή. Στο πανεπιστήμιο του Wisconsin–Madison έχει αναπτυχθεί ένα γενικό runtime σύστημα[2], που χωρίζει τα δεδομένα εισόδου, διαχειρίζεται τις εκτελέσεις των προγραμμάτων πάνω στους διάφορους πυρήνες του Cell, διαχειρίζεται όλες τις μεταφορές των δεδομένων από την τοπική μνήμη των SPEs στην κύρια μνήμη και το αντίθετο. Διαχειρίζεται επίσης και την ενδοεπικοινωνία μεταξύ των πυρήνων και το συγχρονισμό τους. Με τη χρήση του συστήματος αυτού ο χρήστης το μόνο που έχει να κάνει είναι να γράψει τις συναρτήσεις ***map*** και ***reduce.*** Οι λειτουργίες της ***map*** και της ***reduce*** εκτελούνται αποκλειστικά από τους SPEs. Το μοντέλο του mapReduce πάνω στον Cell ακολουθεί μιαν υβριδική προσέγγιση. Δηλαδή, ακολουθεί την προσέγγιση της Google σε clusters (Distributed machines), αφού και ο Cell είναι Distributed memory μηχανή και την προσέγγιση του Phoenix, αφού το granularity των λειτουργιών είναι παρόμοιο. Το μοντέλο αυτό χωρίζεται σε πέντε διαφορετικά στάδια:

Στο πρώτο στάδιο, το ***map*** στάδιο, όπου οι SPEs εκτελούν τη σχεδιασμένη από το χρήστη συνάρτηση ***map*** πάνω στα δεδομένα εισόδου και παράγουν ένα σύνολο κλειδιών και ένα σύνολο τιμών. Τα δεδομένα χωρίζονται σε οχτώ κομμάτια, όσοι είναι και οι επεξεργαστές, και μοιράζονται στα ίσα. Ακολούθως, δεσμεύονται οχτώ buffers για τα δεδομένα εξόδου στην κύρια μνήμη, ένα για κάθε επεξεργαστή. Ο κάθε SPE θα πάρει ένα δείκτη στο buffer του με τα δεδομένα εισόδου και ένα δείκτη στο buffer με τα δεδομένα εξόδου του στην κύρια μνήμη. Στο δεύτερο στάδιο, ***partition***, τα δεδομένα εξόδου από την ***map*** χωρίζονται σε διαφορετικά κομμάτια βάση του κλειδιού που παράχθηκε. Την δουλεία αυτή την αναλαμβάνει ο PPE. Κατά το τρίτο στάδιο, ***Quick-sort***, το σύστημα κατανέμει την ταξινόμηση των αποτελεσμάτων στους οχτώ SPEs και μετά τα επιστρέφουν στην κοινή μνήμη. Κατά την εκτέλεση αυτή υπάρχει πιθανότητα, λόγω του μεγάλου όγκου των δεδομένων και του μικρού μεγέθους της τοπικής μνήμης ενός SPE, να μην γίνει σωστή ταξινόμηση σε ένα κομμάτι των δεδομένων. Συνεπώς μετά το τρίτο στάδιο, αν χρειάζεται, τα δεδομένα αυτά πηγαίνουν στο τέταρτο στάδιο, το ***mergesort***, όπου παίρνουμε όλα τα κομμάτια δεδομένων μαζί και τα ταξινομούμε. Η τέταρτη φάση εκτελείται πάνω στους SPEs με τη βοήθεια DMA transfers(Direct Memory Access μεταφορές δεδομένων). Στο πέμπτο και τελευταίο στάδιο, το ***Reduce***, οι SPEs εκτελούν τη συνάρτηση Reduce, που δίνει ο χρήστης. Καθόλη τη διάρκεια της εκτέλεσης αυτής, ο PPE δίνει και παίρνει πληροφορίες από τους SPEs, αφού και ο ίδιος κρατά δείκτες πάνω στα buffers των δεδομένων εξόδου των SPEs. Με τους δείκτες αυτούς, στο τέλος της εκτέλεσης του πέμπτου σταδίου, ο PPE θα δώσει τα αποτελέσματα της Reduce -τελικά αποτελέσματα- στο χρήστη.

2.3 Mars

Το Mars είναι μια υλοποίηση του μοντέλου MapReduce για χρήση πάνω σε επεξεργαστές γραφικών (GPUs). Σκοπός της υλοποίησης είναι να παρέχει στους προγραμματιστές τη δυνατότητα να εκμεταλλευτούν τη μεγάλη δυνατότητα παραλληλισμού, που παρέχουν τα GPUs [5], για την εκτέλεση προγραμμάτων με μεγάλα δεδομένα εισόδου αποδοτικά και εύκολα. Το Mars κρύβει την πολυπλοκότητα προγραμματισμού τωGPUs πίσω από την απλή, για το χρήστη, διεπαφή του MapReduce, αφού και σε αυτή την υλοποίηση ο προγραμματιστής πρέπει να

υλοποιήσει τις συναρτήσεις Map και Reduce με τέτοιο τρόπο, ώστε να λύνουν το πρόβλημα. Το Mars αναλαμβάνει την αυτόματη μετατροπή του κώδικα για να μπορεί να τρέξει παράλληλα και αποδοτικά στο GPU, έτσι που ο χρήστης να μη χρειάζεται να γνωρίζει το API των γραφικών εντολών ή την αρχιτεκτονική GPU.

Κεφάλαιο 3

Hadoop

3.1 Εντολές χρήσης Hadoop 12
3.2 Πώς να γράψετε ένα πρόγραμμα Hadoop 14
3.3 Πώς να τρέξετε ένα πρόγραμμα Hadoop 17

3.1 Εντολές χρήσης Hadoop

Το Hadoop τυγχάνει χειρισμού από εντολές που εισάγονται από τον χρήστη στην γραμμή εντολών του κεντρικού master κόμβου. Σε αυτό το τμήμα παραθέτω τις εντολές που πιθανόν να χρειαστούν για τη λειτουργία του. Όλες οι εντολές που παρατίθενται εκτελούνται από τον κατάλογο εγκατάστασης του Hadoop.

Εντολή για να γίνει format το file system

```
bin/hadoop namenode -format
```

η συγκεκριμένη εντολή πρέπει να εκτελεστεί αμέσως μετά την εγκατάσταση για να αρχικοποιηθεί το namenode. Δεν πρέπει να εκτελεστεί ξανά, γιατί θα χαθούν όλα τα δεδομένα από το file system.

Εντολή για να ξεκινήσει το cluster του Hadoop

```
bin/start-all.sh
```

η συγκεκριμένη εντολή πρέπει να εκτελεστεί ούτως ώστε να αρχίσουν να τρέχουν όλοι οι συστατικοί κόμβοι του Hadoop (namenode, datanode, jobtracker και tasktracker).

Εντολή αντιγραφής δεδομένων από το τοπικό file system στο HDFS

```
bin/hadoop dfs -copyFromLocal [source dir] [destination dir]
```

η εντολή αυτή αντιγράφει το [source dir] από το τοπικό file system της μηχανής μας στο file system του hadoop (HDFS) στον κατάλογο [destination dir].

Εντολή αντιγραφής δεδομένων από το HDFS στο τοπικό file system

```
bin/hadoop dfs -copyToLocal [source dir] [destination dir]
```

η εντολή αυτή αντιγράφει το [source dir] από το HDFS στο τοπικό file system της μηχανής μας στον κατάλογο [destination dir].

Εντολή προβολής περιεχομένων HDFS

```
bin/hadoop dfs -ls
```

η εντολή αυτή μπορεί να χρησιμοποιηθεί για να έχει κανείς πρόσβαση στο περιεχόμενο του HDFS ή αν χρησιμοποιηθεί σε συνδυασμό με το όνομα ενός φακέλου του file system, μπορεί να προβάλει το περιεχόμενο του φακέλου αυτού ακριβώς όπως η εντολή ls των Linux.

Εντολή εκτέλεσης προγράμματος Hadoop

```
bin/hadoop jar [hadoop program] [input] [output]
```

η εντολή αυτή τρέχει το [hadoop program] πάνω στο Hadoop και χρησιμοποιεί σαν είσοδο τον κατάλογο [input] και έξοδο τον κατάλογο [output] και οι δύο κατάλογοι βρίσκονται στο HDFS.

Εντολή προβολής αρχείων

```
bin/hadoop dfs -cat [directory]/[file]
```

η εντολή αυτή μπορεί να χρησιμοποιηθεί για να έχει κανείς πρόσβαση στο περιεχόμενο ενός αρχείου που βρίσκεται μέσα στο HDFS. Έχει ίδια λειτουργία με την εντολή cat των Linux.

Εντολή διαγραφής ένας κατάλογος από το HDFS

```
bin/hadoop dfs –rmr [directory]
```

η συγκεκριμένη εντολή διαγράφει τον κατάλογο [directory] από το HDFS είτε είναι άδειος είτε όχι.

Εντολή για να σταματήσει το cluster του Hadoop

```
bin/stop-all.sh
```

η συγκεκριμένη εντολή πρέπει να εκτελεστεί ώστε να τερματιστεί η λειτουργία των συστατικών κόμβων του Hadoop (namenode, datanode, jobtracker και tasktracker).

Πλήρης οδηγός εντολών παραθέτεται σαν παράρτημα στο Παράρτημα Β.

3.2 Πώς να γράψετε ένα πρόγραμμα Hadoop

Αν και το Hadoop είναι γραμμένο σε Java, τα προγράμματα τα οποία τρέχουμε σε αυτό δεν πρέπει κατά ανάγκη να είναι και αυτά γραμμένα σε Java. Μια εφαρμογή Hadoop μπορεί να είναι γραμμένη εκτός από Java σε γλώσσες όπως Python ή C++ (από την έκδοση 0.14.1 και μετά). Στο παράδειγμα που θα περιγράψω πιο κάτω θα εξηγήσω πώς να γράψουμε το δικό μας πρόγραμμα WordCount, σε γλώσσα Python, το οποίο κάνει χρήση του HadoopStreaming API, για να μας επιτρέψει να περνούμε δεδομένα μεταξύ του mapper και του reducer κάνοντας χρήση του STDIN (standard input) και STDOUT (standard output).

Πρώτα πρέπει να υλοποιήσουμε την μέθοδο map, η οποία θα μετρά τις εμφανίσεις της κάθε λέξης στο σύνολο εισόδου. Η map, η οποία θα φυλαχθεί σε ξεχωριστό αρχείο (π.χ map.py), θα γράφει τα αποτελέσματα της στο STDOUT.

map.py

```
#!/usr/bin/env python

import sys

# input comes from STDIN (standard input)

for line in sys.stdin:

    # remove leading and trailing whitespace

    line = line.strip()

    # split the line into words

    words = line.split()

    # increase counters

    for word in words:

        # write the results to STDOUT (standard output);

        # what we output here will be the input for the

        # Reduce step, i.e. the input for reducer.py

        # tab-delimited; the trivial word count is 1

        print '%s\t%s' % (word, 1)
```

Στην συνέχεια θα υλοποιήσουμε τη μέθοδο reduce, η οποία θα αθροίζει την κάθε εμφάνιση της κάθε λέξης και θα παρουσιάζει τα τελικά αποτελέσματα. Η reduce, η οποία θα φυλαχθεί σε ξεχωριστό αρχείο (π.χ reduce.py), θα παίρνει την είσοδο της από το STDIN.

reduce.py

```
#!/usr/bin/env python

from operator import itemgetter

import sys

# maps words to their counts
```

```
word2count = {}

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)
    # convert count (currently a string) to int
    try:
        count = int(count)
        word2count[word] = word2count.get(word, 0) + count
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        pass
# sort the words lexigraphically;
# this step is NOT required, we just do it so that our
# final output will look more like the official Hadoop
# word count examples
sorted_word2count = sorted(word2count.items(), key=itemgetter(0))
# write the results to STDOUT (standard output)
for word, count in sorted_word2count:
    print '%s\t%s'% (word, count)
```

Αφού αποθηκεύσουμε τα αρχεία map.py και reduce.py στον κατάλογο που έχουμε εγκαταστήσει το Hadoop, μπορούμε να δοκιμάσουμε το πρόγραμμα μας με την εντολή

```
bin/hadoop jar contrib/streaming/hadoop-0.19.1-streaming.jar -mapper
mapper.py -reducer reducer.py -input [input directory] -output [output
directory]
```

3.3 Πώς να τρέξετε ένα πρόγραμμα Hadoop

Για να τρέξετε ένα πρόγραμμα Hadoop πρέπει πρώτα να εγκαταστήσετε μια έκδοση του Hadoop σε ένα φάκελο της επιλογής σας. Επειδή όλα τα δεδομένα εισόδου πρέπει να είναι στο HDFS πρέπει να τα αντιγράψετε από το τοπικό file system του λειτουργικού σας. Πρώτα πρέπει να ξεκινήσουμε το Hadoop τρέχοντας, από τον φάκελο που το εγκαταστήσαμε, την εντολή:

```
bin/start-all.sh
```

Αυτή η εντολή θα ξεκινήσει τα Namenode, Datanode, Jobtracker και Tasktracker πάνω στην μηχανή. Με την εντολή:

```
bin/hadoop dfs -copyFromLocal [source dir] [destination dir]
```

αντιγράφετε τα δεδομένα εισόδου στο HDFS. Για να εκτελέσετε το πρόγραμμα τρέξετε την εντολή:

```
bin/hadoop jar [program] [input] [output]
```

Για να μεταφέρετε τα αποτελέσματα από το HDFS στο τοπικό file system του λειτουργικού σας για να μπορείτε να τα διαβάσετε εκτελείτε την εντολή:

```
bin/hadoop dfs –copyToLocal [source dir] [destination dir]
```

Κεφάλαιο 4

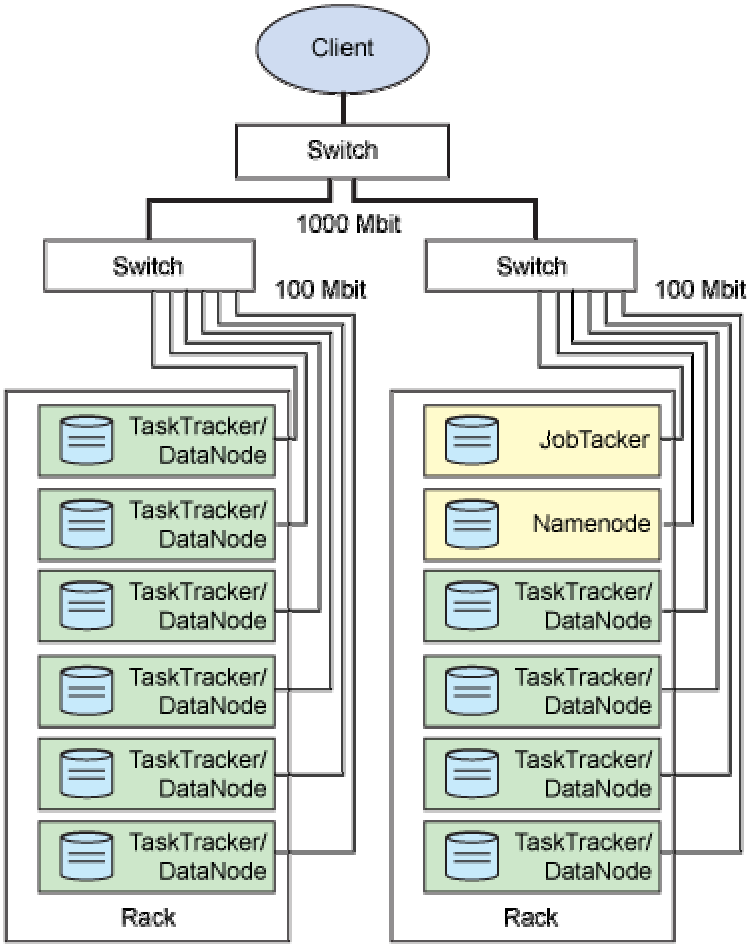
Αρχιτεκτονικές και πλατφόρμες

4.1 Αρχιτεκτονική Hadoop cluster 18
4.2 Αρχιτεκτονική HDFS 21
4.3 Αρχιτεκτονική multicore εγκατάστασης 22
4.4 Multicore επεξεργαστές και παράλληλος προγραμματισμός 23

4.1 Αρχιτεκτονική Hadoop cluster

Ένα δίκτυο εκτέλεσης Hadoop μπορεί να αποτελείται από μια μέχρι και αρκετές δεκάδες χιλιάδες δικτυωμένες μηχανές. Δεν απαιτείται να εγκατασταθεί σε μηχανές συγκεκριμένης αρχιτεκτονικής, λειτουργικού συστήματος ή προδιαγραφών, αφού μπορεί να τρέξει ακόμα και σε οικιακούς υπολογιστές φτάνει να μπορούν να υποστηρίξουν την πλατφόρμα Java.

Μια τυπική μεγάλης κλίμακας εγκατάσταση του Hadoop αποτελείται από τουλάχιστο τις εξής μηχανές κόμβους (nodes): Μια μηχανή που τρέχει τον namenode, μια μηχανή που τρέχει τον jobtracker και αρκετές μηχανές που τρέχουν ένα datanode και ένα tasktracker μαζί (βλέπε σχήμα 2.1). Μπορούμε να επιλέξουμε να εγκαταστήσουμε τα datanodes και τα tasktrackers σε διαφορετικές μηχανές αλλά πρέπει να γίνει με τέτοιο τρόπο, ώστε να μην χάσουμε την βελτιστοποίηση rack awareness, που μας παρέχει το Hadoop, όπου ο jobtracker ξέρει ποιος κόμβος κατέχει τα δεδομένα προς επεξεργασία και ποιοι άλλοι κόμβοι είναι κοντά του και έτσι στέλλει την map ή reduce εργασία σε αυτούς. Αυτή η βελτιστοποίηση βασίζεται στην αρχή του ότι είναι πιο συμφέρον από άποψη χρόνου εκτέλεσης η μετακίνηση της επεξεργασίας παρά η μετακίνηση των δεδομένων.

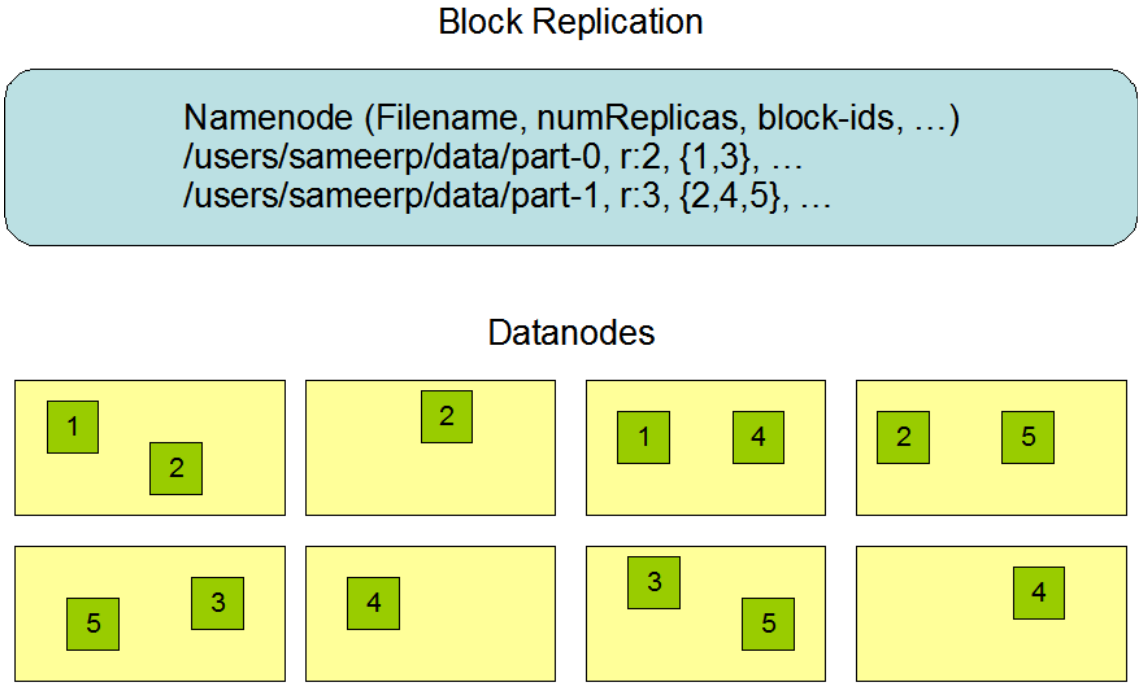


Σχήμα 4.1 Τυπική Αρχιτεκτονική Hadoop

Namenode: αποτελεί τον master κόμβο του HDFS και είναι υπεύθυνος για την διαχείριση του namespace του file system, δηλαδή αναλαμβάνει να κατανέμει τα blocks των δεδομένων, που αποθηκεύονται στο HDFS, σε όλους τους κόμβους του δικτύου και ρυθμίζει την πρόσβαση των client κόμβων στα αρχεία αυτά εκτελώντας εντολές ανοίγματος και κλεισίματος αρχείων. Τέλος, ο namenode είναι υπεύθυνος για την αντιγραφή των δεδομένων και τη διατήρηση αντιγράφων τους σε διάφορους κόμβους του δικτύου για σκοπούς απόδοσης, αλλά και για προστασία από την απώλεια δεδομένων λόγω αστοχίας του υλικού.

Datanodes: αποτελούν τους client κόμβους του HDFS και είναι υπεύθυνοι για την αποθήκευση των block των δεδομένων πάνω στους σκληρούς δίσκους ή σε αλλά αποθηκευτικά μέσα πιθανόν να διαθέτουν οι μηχανές στις οποίες τρέχουν. Οι datanodes εξυπηρετούν αιτήσεις διαβάσματος και γραψίματος από τους jobtracker κόμβους του

δικτύου και μετά από εντολή του namenode δημιουργούν ή διαγράφουν blocks δεδομένων.



Σχήμα 4.2 Διατήρηση αντιγράφων αρχείων

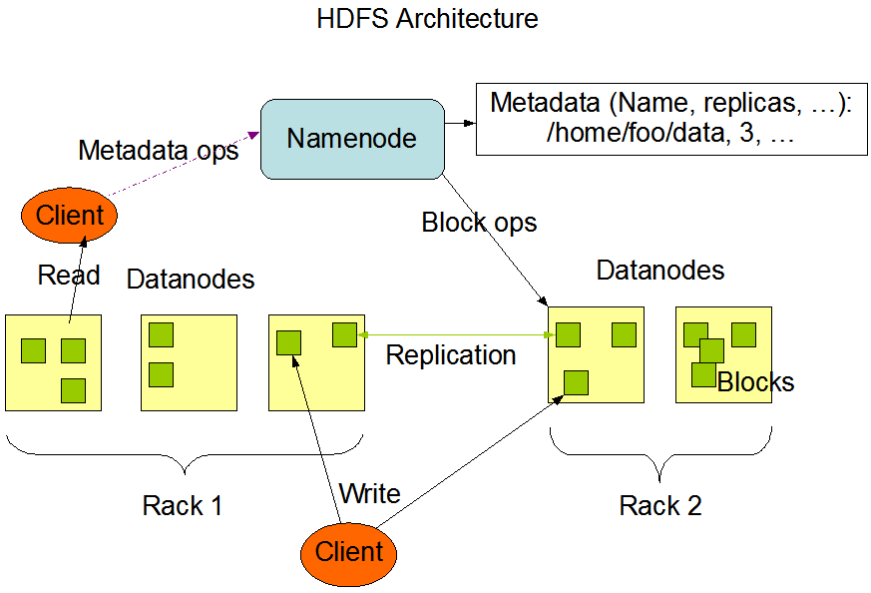
Jobtracker: αποτελεί τον scheduler, που ρυθμίζει την παράλληλη εκτέλεση των εργασιών map και reduce πάνω στο δίκτυο. Ο jobtracker διαθέτει μια στοίβα (stack) στην οποία τοποθετούνται, σε κατάσταση αναμονής, από το Hadoop όλες οι εργασίες που είναι έτοιμες για εκτέλεση. Εφαρμόζοντας την τεχνική FIFO ο jobtracker αναθέτει μια εργασία από την στοίβα του σε όποιο tasktracker είναι διαθέσιμος (idle) για εκτέλεση εφαρμόζοντας ,οπού είναι δυνατό, και την βελτιστοποίηση rack awareness. Για να αρχίσει να μοιράζει τις reduce εργασίες στους κόμβους προς εκτέλεση θα πρέπει πρώτα να τελειώσουν την εκτέλεση τους όλες οι map εργασίες.

Tasktrackers: αποτελούν ουσιαστικά το κομμάτι το οποίο εκτελεί τους υπολογισμούς και διεκπεραιώνει τις map και reduce εργασίες. Αρχίζουν να εκτελούν οποία εργασία τους στείλει ο jobtracker διαβάζοντας τα δεδομένα από το datanode και μετά την επεξεργασία γράφουν τα αποτελέσματα πίσω σε αυτόν. Σε κάθε μηχανή του δικτύου hadoop συνήθως τρέχει ένα μόνο tasktracker και επεξεργάζεται κυρίως τα δεδομένα, που υπάρχουν στο datanode, το οποίο τρέχει στην ίδια με αυτό μηχανή.

4.2 Αρχιτεκτονική HDFS

Το HDFS ακολουθεί το μοντέλο αρχιτεκτονικής client/server και αποτελείται από ένα και μοναδικό namenode τον master, δηλαδή του δικτύου. Ο namenode διαχειρίζεται το namespace του file system και ρυθμίζει την πρόσβαση στα αρχεία από τους clients κόμβους του δικτύου. Εκτός από τον κόμβο namenode υπάρχουν και αρκετοί, συνήθως όσοι και οι κόμβοι του δικτύου Hadoop, datanodes, οι οποίοι χειρίζονται τους σκληρούς δίσκους της μηχανής στην οποία τρέχουν. Αυτή η διαρρύθμιση δίνει στο χρήστη την εντύπωση ότι χρησιμοποίει ένα κοινό και ενιαίο namespace για όλους τους κόμβους του δικτύου πάνω στο οποίο τα δεδομένα μπορούν να αποθηκευτούν / ανακτηθούν από οποιοδήποτε κόμβο του δικτύου.

Εσωτερικά, μέσα στο HDFS, ένα αρχείο σπάζει σε ένα ή περισσότερα κομμάτια (blocks) και αυτά με την σειρά τους αποθηκεύονται σε ένα ή περισσότερα datanodes του δικτύου μας. Ο namenode αναλαμβάνει να καθορίζει την κατανομή (mapping) των blocks μέσα στα datanodes και εκτελεί τις λειτουργίες του namespace, όπως το άνοιγμα / κλείσιμο και η μετονομασία ή μεταφορά αρχείων. Οι datanodes αναλαμβάνουν να εξυπηρετούν τις αιτήσεις διαβάσματος και εγγραφής από τους tasktrackers των κόμβων του Hadoop και μετά από απαίτηση του namenode εκτελούν εντολές δημιουργίας, καταστροφής και αντιγραφής αρχείων.



Σχήμα 4.3 Αρχιτεκτονική HDFS

4.3 Αρχιτεκτονική multicore εγκατάστασης

Για να καταστεί δυνατή η εγκατάσταση του Hadoop πάνω στην multicore μηχανή ήταν απαραίτητο να γίνουν κάποιες αλλαγές πάνω στην αρχιτεκτονική του αλλά και στον τρόπο της λειτουργίας του. Οι εσωτερικοί κόμβοι του Hadoop παρέμειναν οι ίδιοι χωρίς ουσιαστική αλλαγή, επομένως και σε αυτή την εγκατάσταση υπάρχει ο namenode και jobtracker και ένας datanode και tasktracker.

Η μηχανή, που χρησιμοποιήθηκε για την εγκατάσταση, διαθέτει οκτώ επεξεργαστές, από τους οποίους οι τέσσερεις χρησιμοποιήθηκαν για να εγκατασταθούν και να τρέχουν οι namenode, datanode, jobtracker και tasktracker. Σαν αποτέλεσμα παρέμειναν ακόμα τέσσερεις επεξεργαστές για να εκτελούν την παράλληλη εργασία.

Για εκμετάλλευση των πολλών επεξεργαστών της μηχανής ο tasktracker, που υπό τις κανονικές περιστάσεις μπορούσε να αναλάβει μέχρι μια map ή reduce εργασία, έχει τροποποιηθεί έτσι ώστε να μπορεί να αναλάβει περισσότερες. Στα πειράματα που ακολούθησαν μπορούσε να αναλαμβάνει 3, 10, ή και 22 map εργασίες και 1, 2, 3, 5, 9 ή και 15 reduce εργασίες παράλληλα.

Αλλαγές έπρεπε να γίνουν και στον τρόπο που επικοινωνούν οι εσωτερικοί κόμβοι μεταξύ τους. Στην τυπική εγκατάσταση η επικοινωνία γινόταν δυνατή με μηνύματα, που αντάλλαζαν οι μηχανές, οι οποίες έτρεχαν τους κόμβους μέσω του τοπικού δικτύου που τις ένωνε. Στην multicore εγκατάσταση αυτό δεν ήταν δυνατό και ως εκ τούτου για κάθε εσωτερικό κόμβο υιοθετήθηκε ένας ξεχωριστός αριθμός θύρας (port number) και όλα τα μηνύματα επικοινωνίας στέλνονται στην ίδια την μηχανή (localhost) στον αριθμό θύρας του κόμβου που μας ενδιαφέρει.

Τα πλεονεκτήματα της προσέγγισης αυτής είναι το γεγονός ότι όλα τα μηνύματα ανταλλάζονται πολύ πιο γρήγορα, αφού δεν φεύγουν ουσιαστικά από τη μηχανή, και το ότι μπορεί πολύ πιο γρήγορα να εκτελεσθεί η ανάγνωση και εγγραφή δεδομένων, αφού αυτά γίνονται όλα στην ίδια μηχανή.

4.4 Multicore επεξεργαστές και παράλληλος προγραμματισμός

Υπάρχουν αρκετές αρχιτεκτονικές στην περιοχή των CPUs και GPUs. Σχεδόν όλοι οι υπολογιστές, που χρησιμοποιούνται σήμερα, έχουν επεξεργαστές με δυο ή και τέσσερις πυρήνες, ενώ στο μέλλον θα έχουν πολύ περισσότερους. Ακόμα και οι κάρτες γραφικών υψηλών επιδόσεων, που υπάρχουν σε αρκετούς υπολογιστές, διαθέτουν περισσότερους από χίλιους επεξεργαστές.

Δυστυχώς οι παράλληλοι αυτοί επεξεργαστές από μόνοι τους δεν συντελούν στην αύξηση της επίδοσης (μείωση του χρόνου εκτέλεσης) μια εφαρμογής. Τα υπάρχοντα προγράμματα δεν παρέχουν αρκετό παραλληλισμό για να μπορούν να τρέχουν αποδοτικά πάνω σε ένα μεγάλο αριθμό από επεξεργαστές. Για να το πετύχουμε αυτό θα πρέπει να αποκτήσουμε σημαντικές προγραμματιστικές ικανότητες παράλληλου προγραμματισμού.

Ο παράλληλος προγραμματισμός ακολουθεί πολύ διαφορετική φιλοσοφία, και πολλές φορές είναι πιο δύσκολος από το σειριακό προγραμματισμό. Σήμερα υπάρχουν διαθέσιμοι πολλοί παράλληλοι αλγόριθμοι, παράλληλες γλώσσες και προγραμματιστικές τεχνικές. Αυτό που έχουμε να κάνουμε είναι να επιλέξουμε αυτά που μας εξυπηρετούν καλύτερα από άποψη απόδοσης, ευρωστίας και φορητότητας.

Αν και οι παράλληλοι επεξεργαστές υπάρχουν από καιρό στην αγορά, ο παράλληλος προγραμματισμός μόλις τα τελευταία χρόνια άρχισε να χρησιμοποιείται.

Κεφάλαιο 5

Αποτελέσματα πειραμάτων

5.1 Περιβάλλον εγκατάστασης 24
5.2 Τρόπος συλλογής αποτελεσμάτων 24
5.3 Δοκιμαστικές εφαρμογές 26
5.4 Αποτελέσματα multicore εγκατάστασης 29
5.5 Αποτελέσματα cluster εγκατάστασης 37
5.6 Αποτελέσματα εκτέλεσης σε ψευδό-δίκτυο (pseudo-cluster) 39
5.7 Σύγκριση αποτελεσμάτων 40

5.1 Περιβάλλον εγκατάστασης

Για την πραγματοποίηση των πειραμάτων χρησιμοποιήθηκε μια μηχανή IBM ServerX με 2 Intel Xeon E5320, δηλαδή οκτώ πυρήνες/επεξεργαστές, συχνότητας 1,6GHz ο καθένας. Η μηχανή διαθέτει 16GB κύριας μνήμης, από τα οποία μπορούσαν να χρησιμοποιηθούν μόνο τα 3GB, λόγω του ότι το λειτουργικό σύστημα που χρησιμοποιήθηκε(Linux Ubuntu 9.04) ήταν 32bit λόγο καλύτερης συμβατότητας με την έκδοση του Hadoop που χρησιμοποιήθηκε. Ο σκληρός δίσκος της μηχανής είναι χωρητικότητας 500 GB. Από πλευράς λογισμικών χρησιμοποιήθηκε λειτουργικό σύστημα Linux Ubuntu 9.04, η έκδοση του Hadoop ήταν ή v0.19.1 και έτρεχε πάνω από Sun Java 6 (1.6) και τέλος σαν δοκιμαστικές εφαρμογές των πειραμάτων χρησιμοποιήθηκαν οι εφαρμογές WordCount v1.0, η TeraGen και η TeraSort, οι οποίες περιέχονται στο Hadoop σαν δοκιμαστικές εφαρμογές του μοντέλου.

5.2 Τρόπος συλλογής αποτελεσμάτων

Για την συλλογή των αποτελεσμάτων το κάθε σενάριο εκτέλεσης εκτελέστηκε είκοσι φορές και καταγράφηκε ο χρόνος που χρειάστηκε για να τρέξει κάθε εκτέλεση. Αφού ταξινομήθηκαν αυτοί οι χρόνοι σε αύξουσα σειρά, αφαιρέθηκαν η πιο μεγάλη και η πιο μικρή τιμή κάθε σεναρίου. Από τα υπόλοιπα αποτελέσματα υπολογίστηκε ο μέσος όρος του χρόνου εκτέλεσης κάθε σεναρίου και ελέγχθηκαν οι τιμές όλων των εκτελέσεων, έτσι που να μην έχουν απόκλιση από το μέσο όρο, πέραν των πέντε ποσοστιαίων μονάδων (±5%). Να τηρούν δηλαδή τη εξίσωση από το σχήμα 5.1

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

Σχήμα 5.1 Εξίσωση τυπικής απόκλισης

Οι μεγαλύτεροι και οι μικρότεροι χρόνοι εκτέλεσης αφαιρέθηκαν λόγω του γεγονότος ότι πιθανόν κατά την εκτέλεση των σεναρίων κάτι να πήγε στραβά ή κάτι να ενόχλησε την εκτέλεση της εφαρμογής και ο χρόνος αυτούς να μην ήταν αντιπροσωπευτικός του σεναρίου.

Μετά από αυτή την διαδικασία απόμειναν οι αντιπροσωπευτικοί χρόνοι εκτέλεσης κάθε σεναρίου. Όπως φαίνεται και στα σχήματα 5.2, 5.3 και 5.4 δοκιμάστηκαν συνολικά δεκαπέντε σενάρια για την εφαρμογή WordCount, εννέα σενάρια για την εφαρμογή TeraGen και δώδεκα σενάρια για την εφαρμογή TeraSort, των οποίων οι χρόνοι εκτέλεσης θα παρουσιαστούν εκτενέστερα στο κεφάλαιο 5.4 «Αποτελέσματα multicore εγκατάστασης».

	Map = 3	Map = 10	Map = 22
Reduce = 1	x	x	x
Reduce = 2	x	x	x
Reduce = 3	x	x	x
Reduce = 5	x	x	x
Reduce = 9	x	x	x
Reduce = 15	x	x	x

Σχήμα 5.2 Σενάρια Δοκιμών WordCount

	10MB	100MB	1GB
Map = 1	x	x	x
Map = 3	x	x	x
Map = 5	x	x	x

Σχήμα 5.3 Σενάρια Δοκιμών TeraGen

	10MB	100MB	1GB
Map = 1 Reduce = 1	x	x	x
Map = 3 Reduce = 3	x	x	x
Map = 5 Reduce = 3	x	x	x
Map = 7 Reduce = 5	x	x	x

Σχήμα 5.4 Σενάρια Δοκιμών TeraSort

5.3 Δοκιμαστικές εφαρμογές

Για τη διεξαγωγή των πειραμάτων χρησιμοποιήθηκε η εφαρμογή WordCount v1.0. Πρόκειται για μια πολύ απλή και κατανοητή εφαρμογή, η οποία περιέχεται μέσα στο Hadoop, ως παράδειγμα. Ουσιαστικά η εφαρμογή μετρά τις εμφανίσεις κάθε λέξης μέσα σε ένα δοθέν σύνολο εισόδου.

Η λειτουργία της WordCount είναι πολύ απλή. Ο mapper που υλοποιείται στην μέθοδο map (Παράρτημα Α. γραμμές 18-25) επεξεργάζεται μια γραμμή κειμένου κάθε φορά όπως αυτό καθορίζεται από το TextInputFormat (Παράρτημα Α. γραμμή 49). Στην συνέχεια χωρίζει τη γραμμή σε λέξεις χωρίζοντας τις οπού υπάρχει whitespace κάνοντας χρήση της StringTokenizer και επιστρέφει ένα ζεύγος κλειδιού – τιμής για κάθε λέξη < <word>, 1 >

Για παράδειγμα εισόδου τα δυο αρχεία:

Αρχείο 1

```
Hello World Bye World
```

Αρχειο 2

```
Hello Hadoop Goodbye Hadoop
```

Θα εκτελεστούν δυο map μέθοδοι και θα επιστρέψουν η πρώτη

```
< Hello, 1>
< World, 1>
< Bye, 1>
< World, 1>
```

και η δεύτερη

```
< Hello, 1>
< Hadoop, 1>
< Goodbye, 1>
< Hadoop, 1>
```

Η εφαρμογή έχει ένα combiner (Παράρτημα Α. γραμμή 46), ο οποίος κάνει την ίδια δουλεία με τη μέθοδο reduce, αλλά στα τοπικά δεδομένα κάθε map μεθόδου, αφού πρώτα ταξινομηθούν τα αποτελέσματα.

Έτσι τα τελικά ενδιάμεσα αποτελέσματα είναι από την πρώτη map

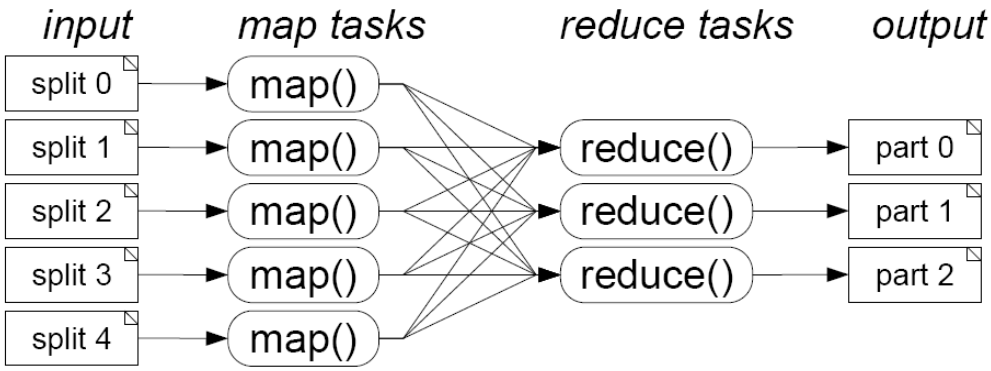
```
< Bye, 1>
< Hello, 1>
< World, 2>
```

Και από την δεύτερη

```
< Goodbye, 1>
< Hadoop, 2>
< Hello, 1>
```

Ο reducer υλοποιείται με τη reduce μέθοδο (Παράρτημα Α. γραμμές 29 -35) και απλά προσθέτει τις τιμές κάθε κλειδιού. Η έξοδος της reduce και κατά συνέπεια της εφαρμογής είναι:

```
< Bye, 1>
< Goodbye, 1>
< Hadoop, 2>
< Hello, 2>
< World, 2>
```



Σχήμα 5.5 Παράδειγμα εκτέλεσης Hadoop

5.4 Αποτελέσματα multicore εγκατάστασης

Σε αυτό το τμήμα παρουσιάζονται τα αποτελέσματα που προέκυψαν από την εκτέλεση των πειραμάτων.

5.4.1 WordCount

Σε όλα τα πειράματα χρησιμοποιήθηκε το ίδιο σύνολο εισόδου (100MB) και τα σενάρια δοκιμάστηκαν με την ίδια εφαρμογή (WordCount).

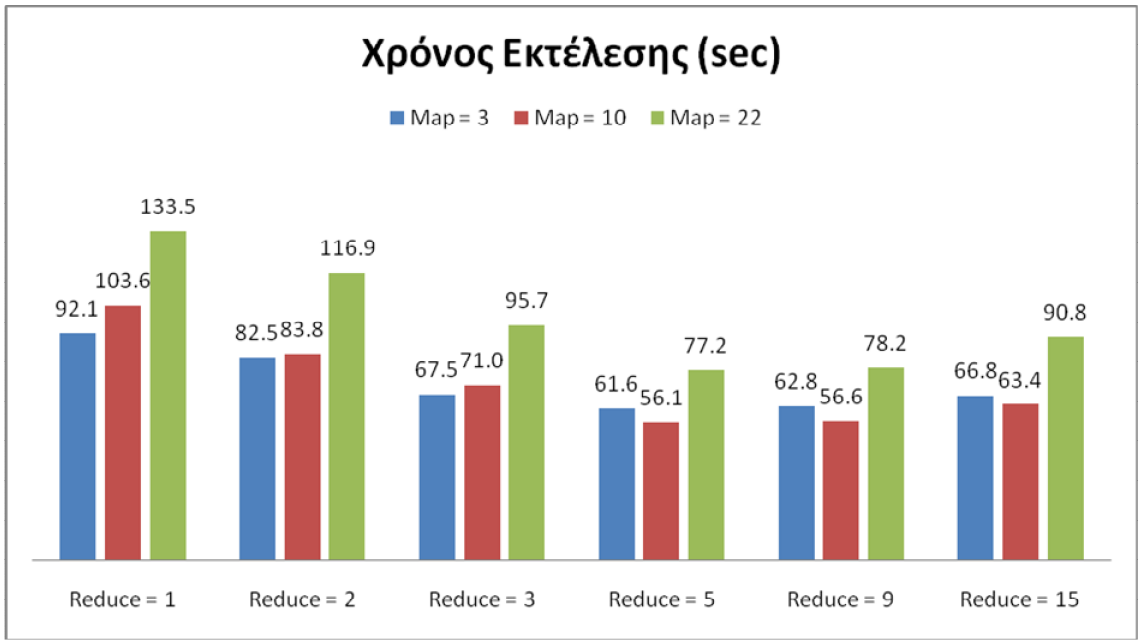
Όλοι οι χρόνοι είναι σε δευτερόλεπτα και τα σενάρια έτρεξαν στο περιβάλλον εγκατάστασης που περιγράφεται στο κεφάλαιο 5.1 «Περιβάλλον εγκατάστασης».

Χρόνοι εκτέλεσης (sec)	Map = 3	Map = 10	Map = 22
Reduce = 1	92.1	103.6	133.5
Reduce = 2	82.5	83.8	116.9
Reduce = 3	67.5	71.0	95.7
Reduce = 5	61.6	56.1	77.2
Reduce = 9	62.8	56.6	78.2
Reduce = 15	66.8	63.4	90.8

Πίνακας 5.6 Αναλυτικοί χρόνοι εκτέλεσης σεναρίων

Στον Πίνακα 5.6 παρουσιάζονται αριθμητικά σε δευτερόλεπτα τα αποτελέσματα (χρόνος εκτέλεσης) των δοκιμών, που έχουν πραγματοποιηθεί για την αξιολόγηση της επίδοσης του Hadoop πάνω στο σύστημα πολυεπεξεργαστών (multicore) για την εφαρμογή WordCount.

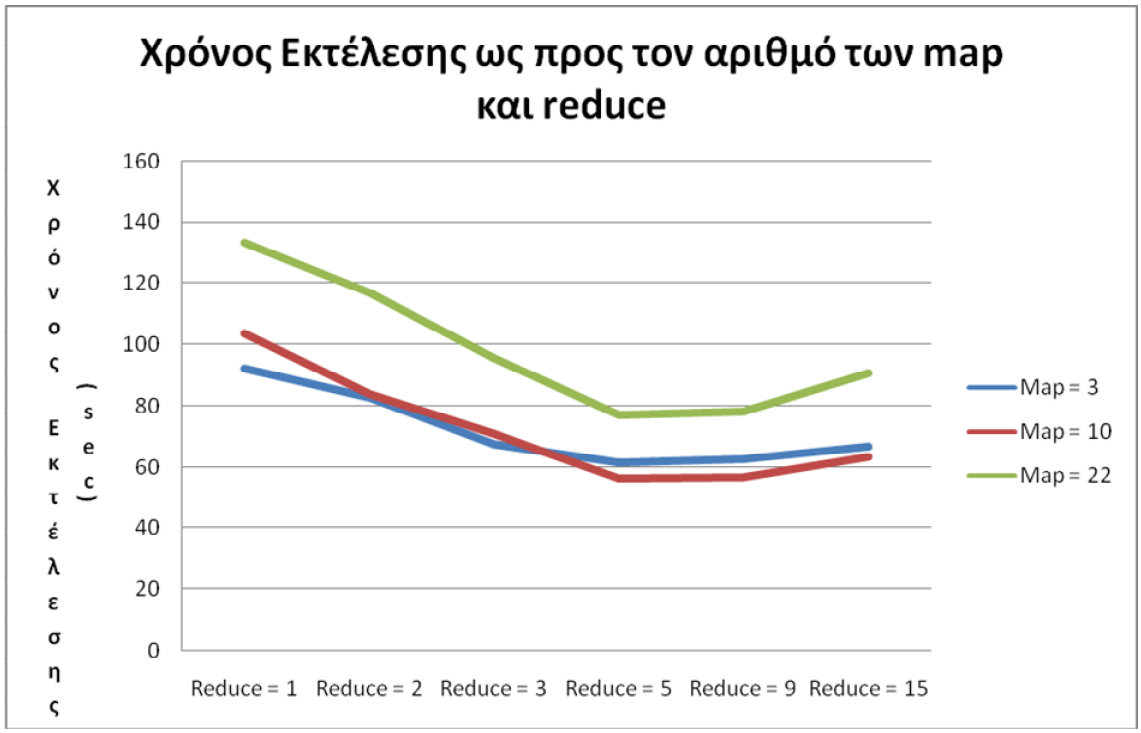
Στη γραφική παράσταση του Σχήματος 5.7 φαίνεται με γραφική αναπαράσταση τους χρόνους εκτέλεσης των διάφορων σεναρίων που δοκιμάστηκαν. Είναι αντιληπτό ότι το σενάριο με map = 10 και reduce = 5 έχει τον καλύτερο χρόνο εκτέλεσης (56.1 sec) από όλα τα σενάρια και το σενάριο με map = 3 και reduce = 1 έχει την χειρότερη επίδοση (133.5 sec).



Σχήμα 5.7 Χρόνος εκτέλεσης σε σχέση με το σενάριο.

Τα 15 σενάρια έχουν επιλεγεί ούτως ώστε να είναι αντιπροσωπευτικά εκτέλεσης σε μικρό αριθμό επεξεργαστών (map = 3 και reduce = 1) και σε μεγάλο αριθμό επεξεργαστών (map = 22 και reduce = 15). Είναι αναγκαίο να τονιστεί ότι η reduce φάση είναι πολύ πιο απαιτητική σε χρόνο εκτέλεσης από τη map φάση και ότι θα ήταν προτιμητέο οι reduce εργασίες να είναι περιττού πλήθους, ώστε να δημιουργείται ένα αντιστραμμένο δέντρο κατά την εκτέλεση τους με κατάληξη στην τελευταία reduce εργασία που αθροίζει τα επιμέρους αθροίσματα των υπόλοιπων reduce εργασιών που προηγήθηκαν.

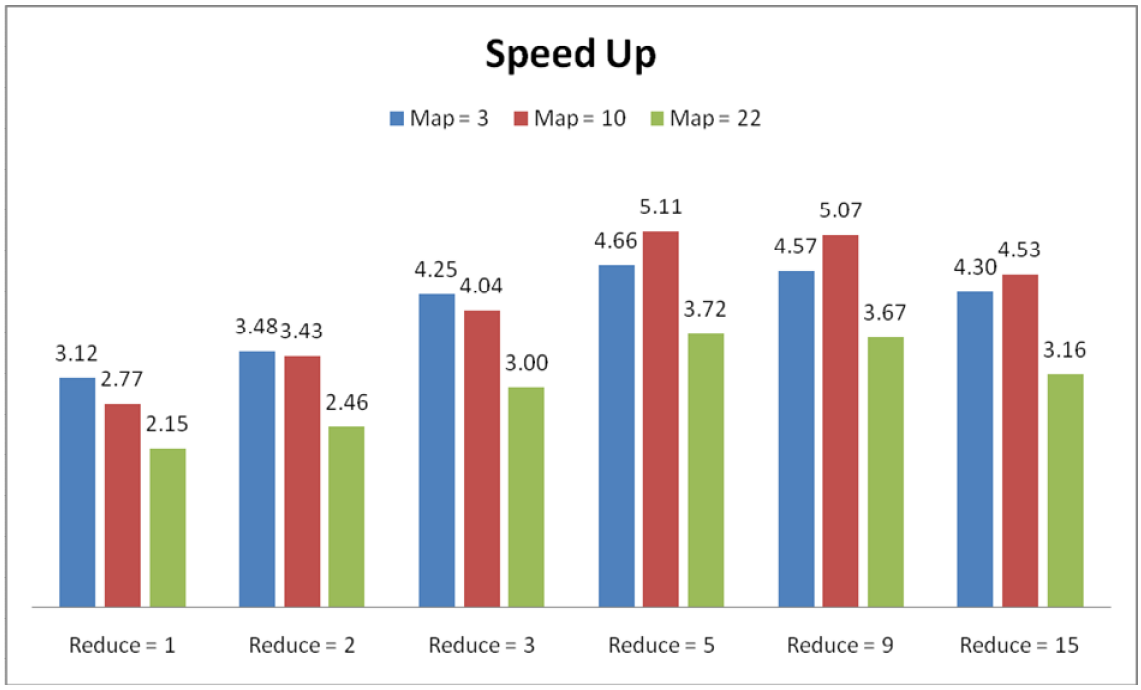
Στο σχήμα 5.8 διαφαίνεται ο τρόπος που αυξομειώνεται ο χρόνος εκτέλεσης σε σχέση με τα σενάρια που έχουν εκλεγεί.



Σχήμα 5.8 Χρόνος εκτέλεσης σε σχέση με το σενάριο

Μπορεί να παρατηρηθεί ότι υπάρχει μια σταδιακή μείωση του χρόνου εκτέλεσης των σεναρίων όσο αυξάνεται ο αριθμός των reduce εργασιών, στις οποίες διασπάτε η reduce φάση της εκτέλεσης. Αυτή η μείωση παρατηρείται μέχρι το σημείο όπου υπάρχουν πέντε reduce και μετά (reduce = 15) αρχίζει να αυξάνεται. Η αύξηση είναι μικρότερη στις περιπτώσεις όπου παρατηρούνται τρείς και δέκα map εργασίες παρά στην περίπτωση με τις είκοσι-δυο reduce εργασίες και αυτό γιατί ο συνολικός αριθμός εργασιών που πρέπει να εκτελέσουν οι επεξεργαστές είναι μικρότερος και συνεπώς μικρότερο είναι και το overhead, που θα υπάρχει στην εκτέλεση από παράγοντες όπως το content switching κτλ.

Στο σχήμα 5.9 φαίνονται τα SpeedUps που έχουν επιτύχει τα σενάρια σε σχέση με την σειριάκη εκτέλεση με map = 1 και reduce = 1.



Σχήμα 5.9 SpeedUp σεναρίων

Η μέγιστη αύξηση της επίδοσης, που έχει επιτευχθεί, είναι 5.11x και η ελάχιστη 2.15x. Τα SpeedUp αυτά είναι αρκετά ικανοποιητικά, αφού, όπως ανάφερα και πιο πάνω, εκτός από τις map και reduce εργασίες, ο επεξεργαστής έχει να τρέχει και τους τέσσερεις κόμβους του Hadoop (namenode, datanode, jobtracker και tasktracker).

5.4.2 TeraGen

Σε όλα τα πειράματα χρησιμοποιήθηκε η ίδια εφαρμογή (TeraGen), η οποία παράγει ένα dataset του οποίου το μέγεθος καθορίζεται από το χρήστη. Ο χρήστης καθορίζει το μέγεθος των γραμμών του dataset, το οποίο είναι της μορφής:

- (10 bytes key) (10 bytes rowid) (78 bytes filler) \r \n
- The keys are random characters from the set ' ' .. '~'.
- The rowid is the right justified row id as a int.
- The filler consists of 7 runs of 10 characters from 'A' to 'Z'.

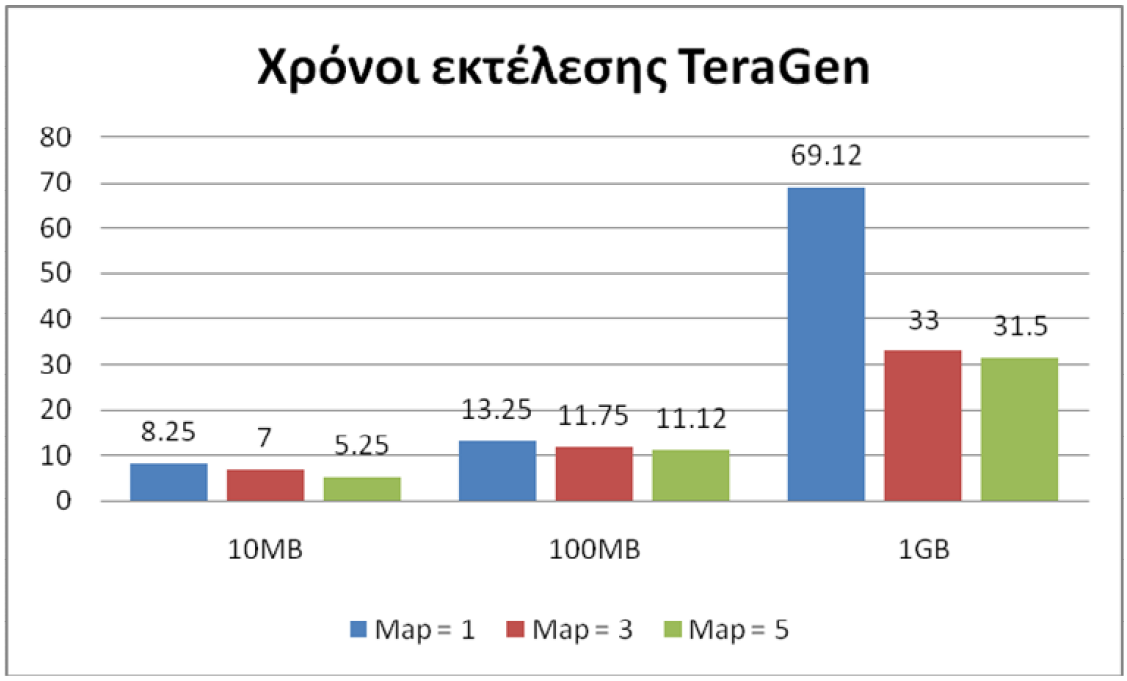
Όλοι οι χρόνοι είναι σε δευτερόλεπτα και τα σενάρια έτρεξαν στο περιβάλλον εγκατάστασης που περιγράφεται στο κεφάλαιο 5.1 «Περιβάλλον εγκατάστασης».

	10MB	100MB	1GB
Map = 1	8.25	13.25	69.12
Map = 3	7	11.75	33
Map = 5	5.25	11.12	31.5

Πίνακας 5.10 Αναλυτικοί χρόνοι εκτέλεσης σεναρίων

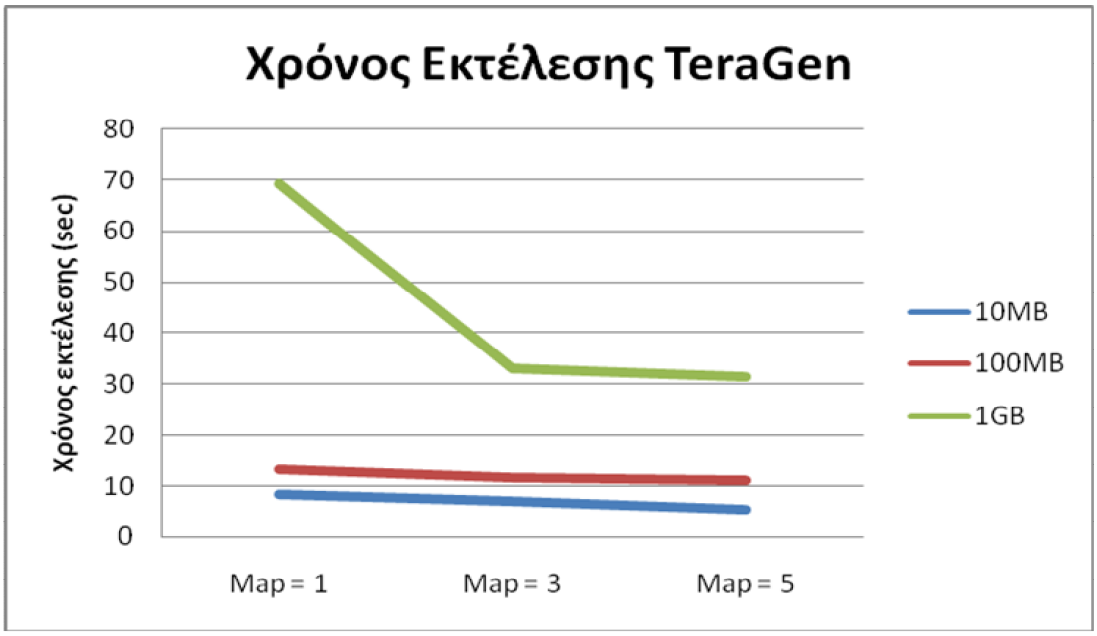
Στον Πίνακα 5.10 παρουσιάζονται αριθμητικά σε δευτερόλεπτα τα αποτελέσματα (χρόνος εκτέλεσης) των δοκιμών, που έχουν πραγματοποιηθεί για την αξιολόγηση της επίδοσης του Hadoop πάνω στο σύστημα πολυεπεξεργαστών (multicore) για την εφαρμογή TeraGen.

Στην γραφική παράσταση του σχήματος 5.11 φαίνονται με γραφική αναπαράσταση οι χρόνοι εκτέλεσης των διάφορων σεναρίων, που δοκιμάστηκαν. Διαφαίνεται ότι τα σενάρια με τις περισσότερες map εργασίες (map = 5) έχουν και τους μικρότερους χρόνους εκτέλεσης.



Σχήμα 5.11 Χρόνος εκτέλεσης σε σχέση με το σενάριο

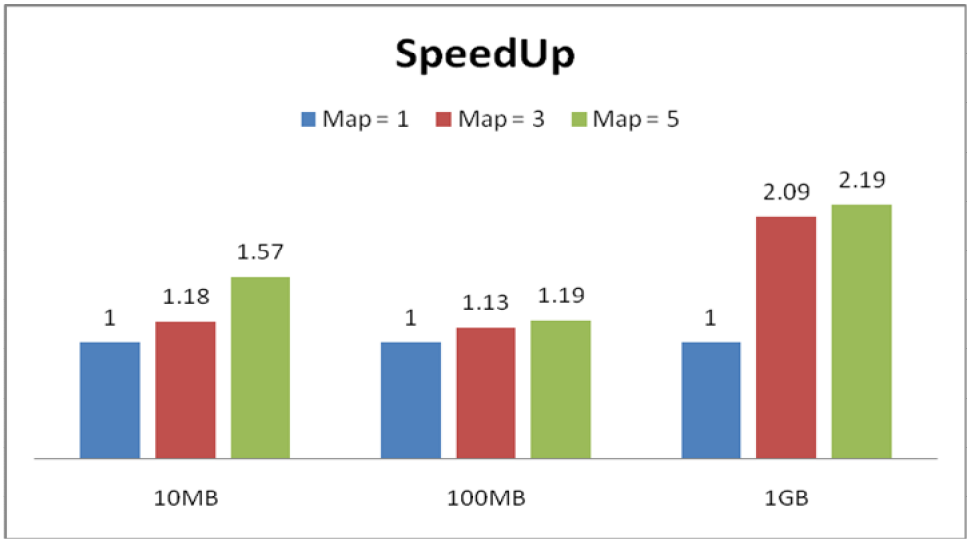
Στο σχήμα 5.12 παρατηρείται ο τρόπος με το οποίο αυξομειώνεται ο χρόνος εκτέλεσης σε σχέση με τα σενάρια που έχουν εκλεγεί.



Σχήμα 5.12 Χρόνος εκτέλεσης σε σχέση με το σενάριο

Φαίνεται ότι υπάρχει μια αρκετά μεγάλη μείωση του χρόνου εκτέλεσης μεταξύ του σεναρίου map = 1 (69.12 sec) και του σεναρίου map = 3 (33 sec), αλλά αυτή η μείωση εξομαλύνεται στο επόμενο σενάριο map = 5 (31.5 sec), όσο αφορά στο σενάριο με το dataset μεγέθους 1GB. Στα άλλα δύο μεγέθη (10MB και 100MB) παρατηρείται πάλι κάποια μείωση αλλά πολύ μικρότερου βαθμού.

Στο σχήμα 5.13 φαίνονται τα SpeedUps που έχουν επιτύχει τα σενάρια σε σχέση με την σειριάκη εκτέλεση με map = 1.



Σχήμα 5.13 SpeedUp σεναρίων

Η μέγιστη αύξηση της επίδοσης που έχει επιτευχθεί είναι 2.19x με ιδανική αύξηση 5x και η ελάχιστη 1.13x με ιδανική αύξηση 3x. Τα SpeedUp αυτά είναι ικανοποιητικά, αφού η εφαρμογή εκτελεί μόνο map εργασίες κατά τις οποίες σπαταλάει περισσότερο χρόνο για να γράφει στον δίσκο της μηχανής παρά για εκτελεί υπολογισμούς.

5.4.3 TeraSort

Σε όλα τα πειράματα χρησιμοποιήθηκε η ίδια εφαρμογή (TeraSort), η οποία παίρνει σαν είσοδο ένα dataset και το επιστρέφει ταξινομημένο. Σαν είσοδος χρησιμοποιήθηκαν dataset, τα οποία παράχθηκαν με την εφαρμογή TeraGen του τμήματος 5.4.2.

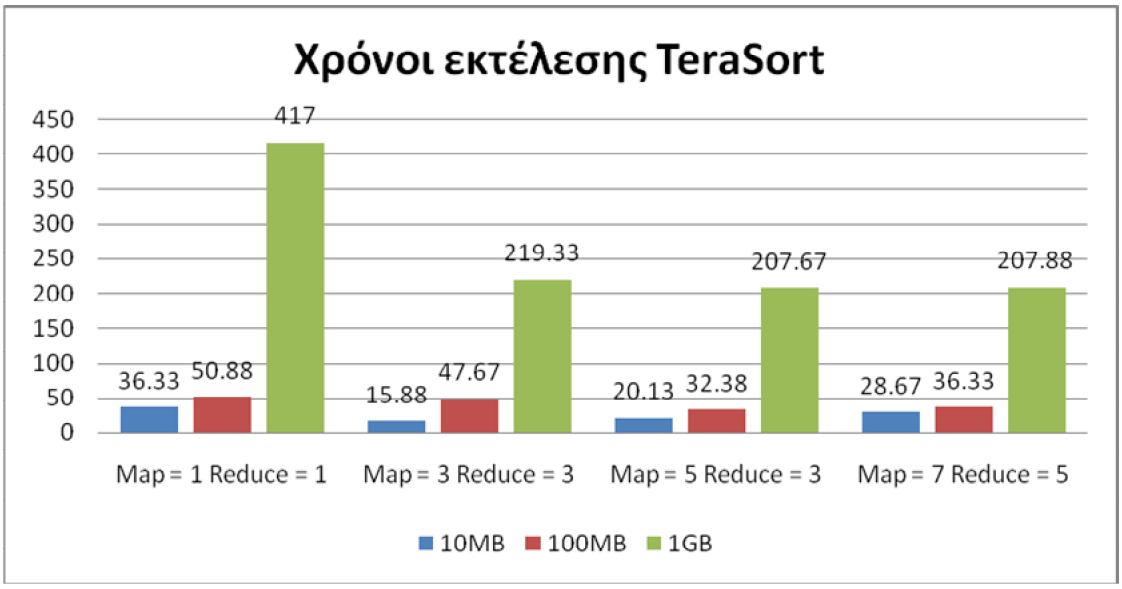
Όλοι οι χρόνοι είναι σε δευτερόλεπτα και τα σενάρια έτρεξαν στο περιβάλλον εγκατάστασης που περιγράφεται στο κεφάλαιο 5.1 «Περιβάλλον εγκατάστασης».

	10MB	100MB	1GB
Map = 1 Reduce = 1	36.33	50.88	417
Map = 3 Reduce = 3	15.88	47.67	219.33
Map = 5 Reduce = 3	20.13	32.38	207.67
Map = 7 Reduce = 5	28.67	36.33	207.88

Πίνακας 5.14 Αναλυτικοί χρόνοι εκτέλεσης σεναρίων

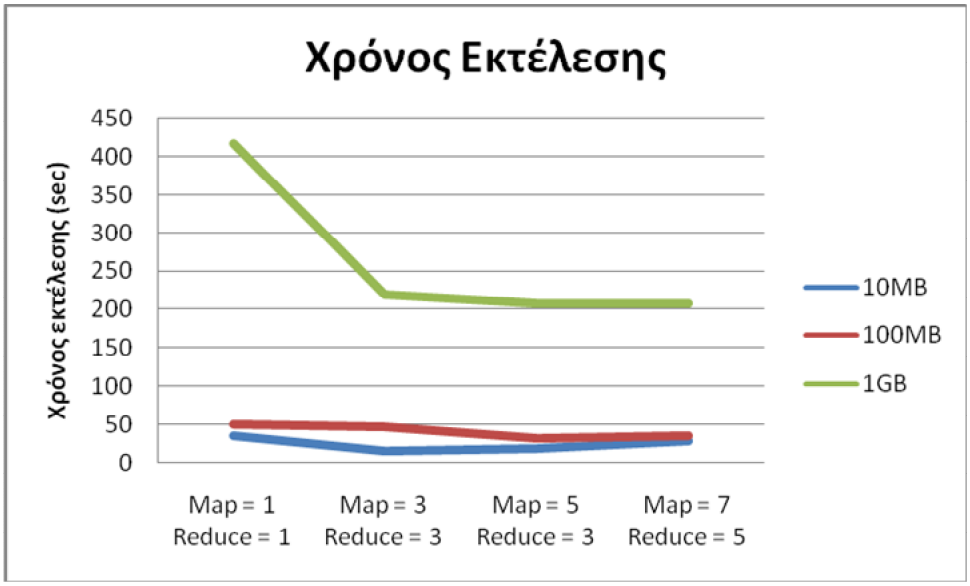
Στον Πίνακα 5.14 παρατίθενται αριθμητικά σε δευτερόλεπτα τα αποτελέσματα (χρόνος εκτέλεσης) των δοκιμών, που έχουν πραγματοποιηθεί για την αξιολόγηση της επίδοσης του Hadoop πάνω στο σύστημα πολυεπεξεργαστών (multicore) για την εφαρμογή TeraSort.

Στην γραφική παράσταση του σχήματος 5.15 φαίνονται με γραφική αναπαράσταση οι χρόνοι εκτέλεσης των διάφορων σεναρίων που δοκιμάστηκαν. Παρατηρείται ότι τα σενάρια με map = 5 και reduce = 3 έχουν τους μικρότερους χρόνους εκτέλεσης.



Σχήμα 5.15 Χρόνος εκτέλεσης σε σχέση με το σενάριο

Στο σχήμα 5.12 διαφαίνεται ο τρόπος με τον οποίο αυξομειώνεται ο χρόνος εκτέλεσης σε σχέση με τα σενάρια που έχουν εκλεγεί.

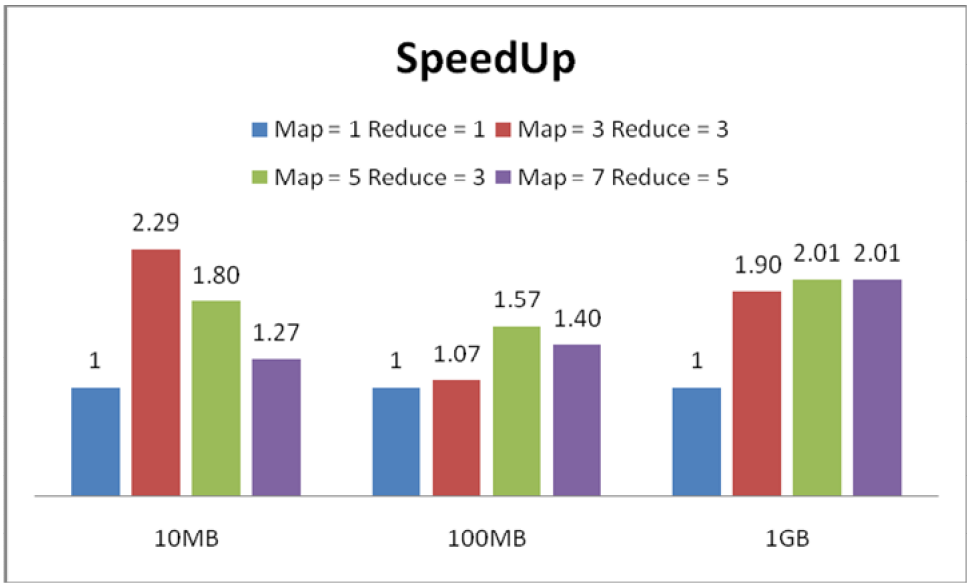


Σχήμα 5.16 Χρόνος εκτέλεσης σε σχέση με το σενάριο

Παρατηρείται να υπάρχει μια αρκετά μεγάλη μείωση του χρόνου εκτέλεσης μεταξύ του σεναρίου map = 1 reduce = 1 (417 sec) και του σεναρίου map = 3 reduce = 3 (219.33 sec), αλλά αυτή η μείωση εξομαλύνεται στα επόμενα σενάρια, στα οποία σχεδόν υπάρχουν οι ίδιοι χρόνοι εκτέλεσης. Διαφαίνεται μια μικρή αύξηση του χρόνου

εκτέλεσης στο τελευταίο σενάριο (map = 7 reduce = 5) που πολύ πιθανό να οφείλεται στα overhead της παράλληλης εκτέλεσης.

Στο σχήμα 5.17 φαίνονται τα SpeedUps που έχουν επιτύχει τα σενάρια σε σχέση με την σειριάκη εκτέλεση με map = 1.



Σχήμα 5.17 SpeedUp σεναρίων

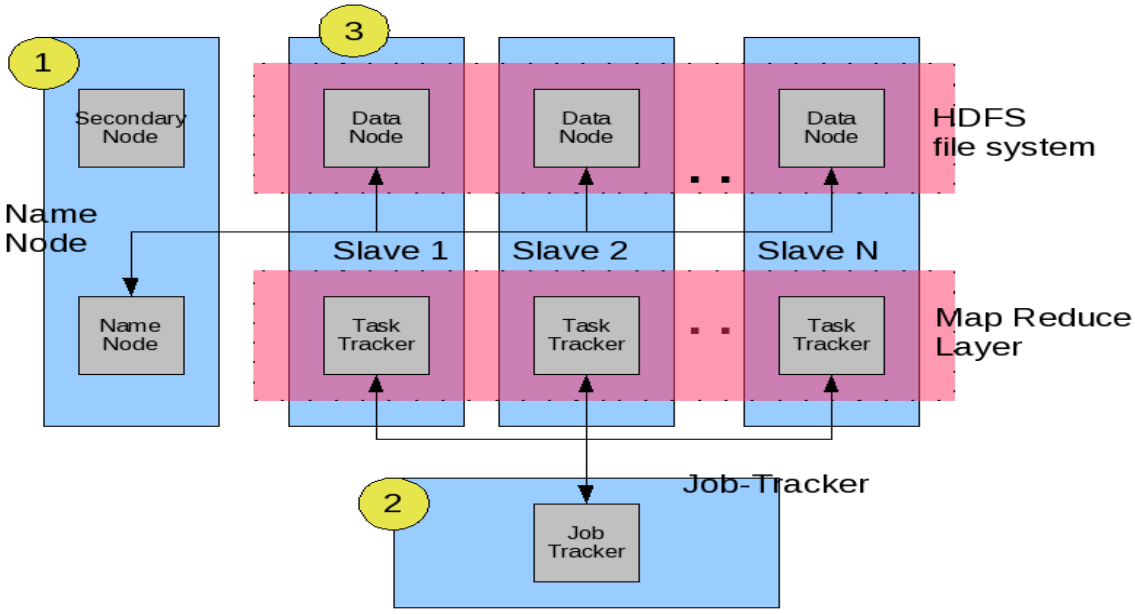
Παρατηρείται αύξηση τις επίδοσης από 1.07x μέχρι και 2.29x. Η επίδοση φαίνεται να περιορίζεται από το μεγάλο όγκο δεδομένων που διαβάζονται και γράφονται από την εφαρμογή μας. Εξ ορισμού η εφαρμογή διαβάζει ολόκληρο το dataset και μετά το γράφει ξανά ταξινομημένο.

5.5 Αποτελέσματα cluster εγκατάστασης

Για σκοπούς σύγκρισης της επίδοσης της multicore εγκατάστασης του Hadoop η δοκιμαστική εφαρμογή WordCount έχει δοκιμαστεί και σε cluster εγκατάσταση του Hadoop με οκτώ μηχανές.

Το cluster αποτελείται από οκτώ μηχανές με επεξεργαστές Intel Celeron, 512 MB κύρια μνήμη (Ram) και 20 GB σκληρό δίσκο. Οι μηχανές τρέχουν λειτουργικό σύστημα Linux Fedora 10 και είναι διασυνδεδεμένες με network switch 10/100/1000 Mbits.

Η εγκατάσταση έγινε όπως στο σχήμα 5.18 με μία μηχανή να εκτελεί τον namenode (1), μια μηχανή να εκτελεί τον jobtracker (2) και έξι μηχανές να εκτελούν από ένα datanode και ένα tasktracker (3). Πρέπει να σημειωθεί ακόμα ότι στην cluster εγκατάσταση ο αριθμός των map και reduce εργασιών, που θα εκτελεστούν, αποφασίζεται αυτόματα από το runtime του Hadoop και έχει να κάνει με τον αριθμό των κόμβων, το μέγεθος αλλά και το πλήθος των αρχείων εισόδου και με το βαθμό αντιγραφής των δεδομένων μέσα στο HDFS.



Σχήμα 5.18 Αρχιτεκτονική Cluster

Στο πιο πάνω cluster έτρεξε η ίδια εφαρμογή με τη multicore εγκατάσταση, (WordCount) με το ίδιο σύνολο εισόδου (100 MB) και εφαρμόστηκε η μεθοδολογία συλλογής αποτελεσμάτων που περιγράφεται στο τμήμα 5.2 «Τρόπος συλλογής αποτελεσμάτων». Το πλήθος των map και reduce εργασιών, που εκτελέστηκαν, αποφασίστηκε από το runtime του Hadoop και ήταν map = 20 και reduce = 5. Τα αποτελέσματα (χρόνος εκτέλεσης), που προέκυψαν από την εκτέλεση, παρουσιάζονται στο σχήμα 5.19 και ο μέσος χρόνος εκτέλεσης είναι 197.39 δευτερόλεπτα.

Εκτέλεση	1	2	3	4	5	6	7	8	9	10
Χρόνος (sec)	163	168	174	176	178	183	183	183	183	185

Εκτέλεση	11	12	13	14	15	16	17	18	19	20
Χρόνος (sec)	186	196	199	206	212	214	235	244	248	263

Σχήμα 5.19 Χρόνοι εκτέλεσης cluster εγκατάστασης

5.6 Αποτελέσματα εκτέλεσης σε ψευδό-δίκτυο (pseudo-cluster)

Το pseudo-cluster είναι ένας τρόπος εγκατάστασης που παρέχεται από το Hadoop για την τοπική εγκατάσταση του σε μια μηχανή για σκοπούς δοκιμής της λειτουργίας του. Για την λειτουργία του pseudo-cluster γίνονται αλλαγές στις ρυθμίσεις του Hadoop παρόμοιες με αυτές που έγιναν για τον σκοπό της εργασίας αυτής και που περιγράφονται στο κεφάλαιο 4.3 Αρχιτεκτονική multicore εγκατάστασης. Λόγω του ότι η εκτέλεση των map και reduce εργασιών γίνεται από τον ίδιο επεξεργαστή η επίδοση αυτής της εγκατάστασης δεν μπορεί να αυξάνεται αλλά να μειώνεται καθώς αυξάνουμε τον αριθμό των εργασιών που θα εκτελεστούν. Αυτό οφείλεται στο επιπλέον overhead που προστίθεται στην εκτέλεση λόγο της εναλλαγής των εργασιών στον επεξεργαστή για εκτέλεση. Τα αποτελέσματα της εκτέλεσης της εφαρμογής WordCount με 100MB δεδομένα εισόδου παρουσιάζονται πιο κάτω.

	Map = 3	Map = 10	Map = 22
Reduce = 1	257.80	275.30	286.30
Reduce = 2	272.20	301.50	310.20
Reduce = 3	303.70	318.20	325.30
Reduce = 5	306.60	322.60	340.10
Reduce = 9	311.20	350.10	342.90
Reduce = 15	316.40	353.70	359.30

Σχήμα 5.20 Χρόνοι εκτέλεσης σε δευτερόλεπτα.

5.7 Σύγκριση αποτελεσμάτων

Για τη σύγκριση των δυο υλοποιήσεων cluster και multicore χρησιμοποιήθηκαν οι χρόνοι εκτέλεσης της εφαρμογής WordCount σε cluster και οι χρόνοι εκτέλεσης του σεναρίου map =22 reduce = 5 από την multicore εγκατάσταση, αφού είναι το πιο κοντινό σενάριο στις map και reduce εργασίες, που έτρεξε το runtime του Hadoop πάνω στο cluster (map = 20 reduce = 5).

Στο σχήμα 5.20 φαίνονται οι χρόνοι των δύο εγκαταστάσεων. Παρατηρείται ότι ο χρόνος εκτέλεσης στην multicore εγκατάσταση είναι κατά 60% καλύτερος από αυτόν του cluster.

Multicore	77.20
Cluster	127.10

Σχήμα 5.21 Χρόνοι εκτέλεσης σε δευτερόλεπτα.

Λόγω του ότι οι επεξεργαστές που έτρεχαν στο cluster έχουν διαφορετική συχνότητα από τους πυρήνες του multicore συστήματος οι πιο πάνω χρόνοι έχουν υπολογιστεί μετά από μια διαδικασία εξομάλυνσης της ανομοιομορφίας της συχνότητας. Υπολογίστηκε δηλαδή ο χρόνος που χρειάζεται για να μεταφερθούν τα δεδομένα μέσο του τοπικού δικτύου στην περίπτωση του cluster και αυτός ο χρόνος αφαιρέθηκε από τον συνολικό χρόνο εκτέλεσης. Στην συνέχεια διαιρέθηκε ο υπόλοιπος χρόνος εκτέλεσης του cluster με την κλίμακα της διαφοράς των συχνοτήτων των επεξεργαστών έτσι που να εξομαλυνθεί η ανομοιομορφία. Στον καινούργιο χρόνο εκτέλεσης προστέθηκε ο χρόνος της μεταφοράς των δεδομένων μέσο του δικτύου.

Τα πιο πάνω αποτελέσματα είναι αναμενόμενα, γιατί στην multicore εγκατάσταση δεν υπάρχει το επιπλέον κόστος σε χρόνο της μετακίνησης των δεδομένων από κόμβο σε κόμβο, αφού όλα βρίσκονται στην ίδια κοινή μνήμη (σκληρός δίσκος) .

Τα αποτελέσματα από την εκτέλεση όλων των εφαρμογών μας δείχνουν ότι το Hadoop μας δίνει speedup όταν τρέχει παράλληλα πάνω σε multicore επεξεργαστή και πιο

συγκεκριμένα το speedup αυτό είναι μεγαλύτερο όταν ο αριθμός των εργασιών που θα εκτελεστούν είναι πολλαπλάσιος του αριθμού τον tasktrackers που έχουμε στο σύστημα μας.

Κεφάλαιο 6

Συμπεράσματα και μελλοντική εργασία

6.1 Συμπεράσματα 42
6.2 Μελλοντική Εργασία 44

6.1 Συμπεράσματα

Το Hadoop αλλά και γενικότερα το προγραμματιστικό μοντέλο MapReduce έχει αποδειχτεί να είναι αρκετά αποτελεσματικό και αποδοτικό στην παράλληλη εκτέλεση αρκετών και διάφορων εργασιών. Αποδίδω αυτή την επιτυχία σε διάφορους λόγους:

- Καταρχάς το μοντέλο είναι πολύ εύκολο στη χρήση του ακόμα και για προγραμματιστές με λίγη ή και καθόλου εμπειρία στον παράλληλο προγραμματισμό, αφού κρύβει τις λεπτομέρειες του παραλληλισμού, της ανάκαμψης από σφάλματα, των locality optimizations και του load balancing. Ουσιαστικά ο προγραμματιστής αυτό που έχει να κάνει, για να τρέξει το πρόγραμμα του με το Hadoop, είναι να υλοποιήσει τις δυο συναρτήσεις map και reduce με τέτοιο τρόπο ώστε να λύνουν το πρόβλημα του και να αφήσει τα υπόλοιπα στο Hadoop.

- Ένα μεγάλο εύρος προβλημάτων μπορούν σχετικά εύκολα να εκφραστούν σαν MapReduce υπολογισμοί. Για παράδειγμα το μοντέλο αυτό μπορεί να εφαρμοστεί σε μηχανές αναζήτησης, για ταξινόμηση, για data mining, για machine learning και αρκετές άλλες εφαρμογές.

- Το Hadoop μπορεί εύκολα να γίνει scale για χρήση σε εκατοντάδες ή και χιλιάδες επεξεργαστές με τρόπο που να εκμεταλλεύεται αποδοτικά τους πόρους

των μηχανών πάνω στις οποίες τρέχει, διατηρώντας για τον προγραμματιστή την ίδια ευκολία εκτέλεσης με τη σειριακή εκτέλεση σε ένα και μοναδικό υπολογιστή.

Με μικρές σχετικά αλλαγές το Hadoop μπορεί να ρυθμιστεί, ώστε να τρέχει αποδοτικά και πάνω σε συστήματα πολυεπεξεργαστών, αντί πάνω σε clusters. Η προσέγγιση αυτή βοηθά στην επίλυση μερικών από τα bottle necks του Hadoop. Τα προβλήματα αυτά, που επιλύει η multicore εγκατάσταση είναι:

- Το περιορισμένο bandwidth της διασύνδεσης των κόμβων του cluster μεταξύ τους επηρεάζει σημαντικά την επίδοση, αφού είναι αρκετά μεγάλος ο όγκος των δεδομένων που διακινούνται μέσω του δικτύου. Οι βελτιστοποιήσεις, που παρέχει το Hadoop, η τοπικότητα της εκτέλεσης και η μεταφορά υπολογισμού παρά δεδομένων, βοηθούν σε αρκετό βαθμό, αλλά στο περιβάλλον της multicore εγκατάστασης το πρόβλημα ουσιαστικά εξαλείφεται για το λόγο ότι δεν υπάρχει μεταφορά δεδομένων από και προς άλλες μηχανές.

- Η τοπικότητα των δεδομένων είναι ένα άλλο πρόβλημα επίδοσης που προσπαθεί με τις βελτιστοποιήσεις του να λύσει το Hadoop. Πρόβλημα που και πάλι εξαλείφεται στην multicore εγκατάσταση, αφού όλα τα δεδομένα βρίσκονται στην ίδια μηχανή όπου οι πόροι, μαζί και η μνήμη, μοιράζονται από όλους τους επεξεργαστές πράγμα που από μόνο του προκαλεί μικρότερους χρόνους ανάγνωσης και εγγραφής δεδομένων.

Με την επίλυση αυτών των προβλημάτων στη multicore εγκατάσταση σημειώνεται αύξηση της επίδοσης των προγραμμάτων μέχρι και 5.11x σε επεξεργαστή με οκτώ πυρήνες και χρόνους εκτέλεσης μέχρι και 60% καλύτερος από την εκτέλεση της ίδιας εφαρμογής πάνω σε ένα Hadoop cluster με τα ίδια δεδομένα εισόδου. Η αύξηση αυτή είναι αρκετά ικανοποιητική, αφού εκτός από το να τρέξει το πρόγραμμα, ο επεξεργαστής πρέπει να τρέχει και το Hadoop, το οποίο αποτελείται από τέσσερεις κόμβους (namenode, datanode, jobtracker, tasktracker).

Στα επόμενα χρόνια, όπου αναμένεται να αυξηθούν κατά πολύ οι διαθέσιμοι πυρήνες ανά μηχανή, το Hadoop θα είναι ίσως ένα από τα μοντέλα υπολογισμού που θα

επικρατήσουν στον τομέα της παράλληλης εκτέλεσης. Ακόμα τα σημερινά Hadoop clusters ίσως στο μέλλον να μπορούν να αντικατασταθούν με cluster από multicore μηχανές δίνοντας μας τα πλεονεκτήματα και των δύο υλοποιήσεων. Σε ένα τέτοιο cluster θα μπορούμε να εκμεταλλευτούμε τόσο την ευκολία του scalability του cluster αλλά και την καλύτερη επίδοση της multicore εγκατάστασης.

6.2 Μελλοντική Εργασία

Ασφαλώς, η επιτυχία της διπλωματικής αυτής εργασίας, δεν συνεπάγει και σίγουρα δε διασφαλίζει ότι το σύστημα δεν μπορεί να τύχει των οποιοδήποτε αλλαγών και βελτιώσεων. Αφενός αποδείχθηκε ότι μπορεί να χρησιμοποιηθεί ένα multicore σύστημα για επιτάχυνση της εκτέλεσης εφαρμογών Hadoop, αφετέρου όμως μπορεί να χρησιμοποιηθεί ως έναυσμα περισσότερης ερευνητικής μελέτης και κυρίως στο επίπεδο της κατανάλωσης ηλεκτρικής ενέργειας με σκοπό την βελτίωση της απόδοσης του συνολικού συστήματος με την δημιουργία clusters από multicores.

Η διπλωματική αυτή εργασία, αν και κρίνεται πλήρης όσον αφορά την κάλυψη του θέματος από τον συγγραφέα, εν τούτοις υπάρχουν κάποιοι παράγοντες που δεν ενσωματώθηκαν στο σύστημα και ίσως χρήζουν έρευνας από μελλοντικούς πιθανούς συνεχιστές του έργου. Παράγοντες όπως μια οικονομική μελέτη από πλευράς κόστους εγκατάστασης αλλά και λειτουργίας clusters τα οποία θα λειτουργούν σύμφωνα με τα ευρήματα αυτής της ερευνάς ή ακόμα και αλλαγές στην αρχιτεκτονική που προτάθηκε και διεξαγωγή περισσοτέρων πειραμάτων για ακόμα καλύτερη επίδοση του συστήματος.

Η πιθανή μελλοντική εργασία επικεντρώνεται κατά κύριο λόγο στην υλοποίηση και την διεξαγωγή πειραμάτων σε clusters με multicore μηχανές και την σύγκριση της επίδοσης τους με τα cluster που χρησιμοποιούνται σήμερα. Ακόμα μπορούσε να διεξαχθεί και μια έρευνα κόστους για να διαφανεί αν είναι οικονομικά συμφέρον η εγκατάσταση των cluster με multicore αλλά και αν καταναλώνουν λιγότερη ηλεκτρική ενέργεια για την λειτουργία τους. Επίσης θα μπορούσε να μελετηθεί και η πιθανότητα να γίνει χρήση του Hadoop πάνω σε ετερογενή συστήματα με περισσότερα από ένα είδη επεξεργαστές έτσι που να μπορούν να αξιοποιηθούν τα θετικά στοιχεία περισσότερων επεξεργαστών.

Βιβλιογραφία

[1] Kozyrakis, Christos, et al. Evaluating MapReduce for Multicore and Multiprocessor Systems ,Computer Systems Laboratory Stanford University, 2007.

[2] Sankaralingam, Karthikeyan and Marc de Kruijf MapReduce for the Cell B.E. Architecture, Vertical Research Group Department of Computer Sciences, University of Wisconsin–Madison, 2007.

[3] Chao, Jin and Rajkumar Buyya, MapReduce Programming Model for .NET-based Distributed Computing, Australia, Grid Computing and Distributed Systems (GRIDS) Laboratory. Department of Computer Science and Software Engineering. The University of Melbourne, 2008.

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

[5] He, Bingsheng, et al Mars: A MapReduce Framework on Graphics Processors**,** Hong Kong Univ. of Sci. & Tech. Microsoft Corp. Sina Corp, 2008.

[6] Yeung, Jackson, et al, Map-reduce as a Programming Model for Custom Computing Machines, Hong Kong, Dept. of Computer Science and Engineering
The Chinese University of Hong Kong

[7] “Welcome to Apache Hadoop!”, May 2009, <http://hadoop.apache.org>

Παράρτημα Α

Κώδικας εφαρμογής wordcount v1.0

WordCount.java	
1.	`package org.myorg;`
2.	
3.	`import java.io.IOException;`
4.	`import java.util.*;`
5.	
6.	`import org.apache.hadoop.fs.Path;`
7.	`import org.apache.hadoop.conf.*;`
8.	`import org.apache.hadoop.io.*;`
9.	`import org.apache.hadoop.mapred.*;`
10.	`import org.apache.hadoop.util.*;`
11.	
12.	`public class WordCount {`
13.	
14.	`  public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {`
15.	`    private final static IntWritable one = new IntWritable(1);`
16.	`    private Text word = new Text();`
17.	

```
18.     public void map(LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
19.       String line = value.toString();
20.       StringTokenizer tokenizer = new StringTokenizer(line);
21.       while (tokenizer.hasMoreTokens()) {
22.         word.set(tokenizer.nextToken());
23.         output.collect(word, one);
24.       }
25.     }
26.   }
27.
28.   public static class Reduce extends MapReduceBase implements
      Reducer<Text, IntWritable, Text, IntWritable> {
29.     public void reduce(Text key, Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
30.       int sum = 0;
31.       while (values.hasNext()) {
32.         sum += values.next().get();
33.       }
34.       output.collect(key, new IntWritable(sum));
35.     }
36.   }
37.
```

```
38.   public static void main(String[] args) throws Exception {
39.     JobConf conf = new JobConf(WordCount.class);
40.     conf.setJobName("wordcount");
41.
42.     conf.setOutputKeyClass(Text.class);
43.     conf.setOutputValueClass(IntWritable.class);
44.
45.     conf.setMapperClass(Map.class);
46.     conf.setCombinerClass(Reduce.class);
47.     conf.setReducerClass(Reduce.class);
48.
49.     conf.setInputFormat(TextInputFormat.class);
50.     conf.setOutputFormat(TextOutputFormat.class);
51.
52.     FileInputFormat.setInputPaths(conf, new Path(args[0]));
53.     FileOutputFormat.setOutputPath(conf, new Path(args[1]));
54.
55.     JobClient.runJob(conf);
57.   }
58.}
59.
```

Παράρτημα Β

Οδηγός Χρήσης Hadoop

Overview

All hadoop commands are invoked by the bin/hadoop script. Running the hadoop script without any arguments prints the description for all commands.

```
Usage: hadoop [--config confdir] [COMMAND] [GENERIC_OPTIONS]
[COMMAND_OPTIONS]
```

Hadoop has an option parsing framework that employs parsing generic options as well as running classes.

COMMAND_OPTION	Description
`--config confdir`	**Overwrites the default Configuration directory. Default is ${HADOOP_HOME}/conf.**
`GENERIC_OPTIONS`	**The common set of options supported by multiple commands.**
`COMMAND` `COMMAND_OPTIONS`	**Various commands with their options are described in the following sections. The commands have been grouped into User Commands and Administration Commands.**

Generic Options

The following options are supported by dfsadmin, fs, fsck and job. Applications should implement Tool to support GenericOptions.

GENERIC_OPTION	Description
`-conf <configuration file>`	**Specify an application**

	configuration file.
`-D <property=value>`	**Use value for given property.**
`-fs <local|namenode:port>`	**Specify a namenode.**
`-jt <local|jobtracker:port>`	**Specify a job tracker. Applies only to job.**
`-files <comma separated list of files>`	**Specify comma separated files to be copied to the map reduce cluster. Applies only to job.**
`-libjars <comma seperated list of jars>`	**Specify comma separated jar files to include in the classpath. Applies only to job.**
`-archives <comma separated list of archives>`	**Specify comma separated archives to be unarchived on the compute machines. Applies only to job.**

User Commands

Commands useful for users of a hadoop cluster.

archive

Creates a hadoop archive. More information can be found at Hadoop Archives.

`Usage: hadoop archive -archiveName NAME <src>* <dest>`

COMMAND_OPTION	Description
`-archiveName NAME`	**Name of the archive to be created.**
`src`	**Filesystem pathnames which work as usual with**

	regular expressions.
`dest`	**Destination directory which would contain the archive.**

distcp

Copy file or directories recursively. More information can be found at Hadoop DistCp Guide.

`Usage: hadoop distcp <srcurl> <desturl>`

COMMAND_OPTION	Description
`srcurl`	**Source Url**
`desturl`	**Destination Url**

fs

`Usage: hadoop fs [GENERIC_OPTIONS] [COMMAND_OPTIONS]`

Runs a generic filesystem user client.

The various COMMAND_OPTIONS can be found at Hadoop FS Shell Guide.

fsck

Runs a HDFS filesystem checking utility. See Fsck for more info.

`Usage: hadoop fsck [GENERIC_OPTIONS] <path> [-move | -delete | -openforwrite] [-files [-blocks [-locations | -racks]]]`

COMMAND_OPTION	Description
`<path>`	**Start checking from this path.**
`-move`	**Move corrupted files to /lost+found**
`-delete`	**Delete corrupted files.**

-openforwrite	**Print out files opened for write.**
-files	**Print out files being checked.**
-blocks	**Print out block report.**
-locations	**Print out locations for every block.**
-racks	**Print out network topology for data-node locations.**

jar

Runs a jar file. Users can bundle their Map Reduce code in a jar file and execute it using this command.

```
Usage: hadoop jar <jar> [mainClass] args...
```

The streaming jobs are run via this command. Examples can be referred from Streaming examples

Word count example is also run using jar command. It can be referred from Wordcount example

job

Command to interact with Map Reduce Jobs.

```
Usage: hadoop job [GENERIC_OPTIONS] [-submit <job-file>] | [-status <job-id>] | [-counter <job-id> <group-name> <counter-name>] | [-kill <job-id>] | [-events <job-id> <from-event-#> <#-of-events>] | [-history [all] <jobOutputDir>] | [-list [all]] | [-kill-task <task-id>] | [-fail-task <task-id>] | [-set-priority <job-id> <priority>]
```

COMMAND_OPTION	Description
-submit <job-file>	**Submits the job.**
-status <job-id>	**Prints the map and reduce completion percentage and all job counters.**

`-counter <job-id> <group-name> <counter-name>`	**Prints the counter value.**
`-kill <job-id>`	**Kills the job.**
`-events <job-id> <from-event-#> <#-of-events>`	**Prints the events' details received by jobtracker for the given range.**
`-history [all] <jobOutputDir>`	**-history <jobOutputDir> prints job details, failed and killed tip details. More details about the job such as successful tasks and task attempts made for each task can be viewed by specifying the [all] option.**
`-list [all]`	**-list all displays all jobs. -list displays only jobs which are yet to complete.**
`-kill-task <task-id>`	**Kills the task. Killed tasks are NOT counted against failed attempts.**
`-fail-task <task-id>`	**Fails the task. Failed tasks are counted against failed attempts.**
`-set-priority <job-id> <priority>`	**Changes the priority of the job. Allowed priority values are VERY_HIGH, HIGH, NORMAL, LOW, VERY_LOW**

pipes

Runs a pipes job.

```
Usage: hadoop pipes [-conf <path>] [-jobconf <key=value>,
<key=value>, ...] [-input <path>] [-output <path>] [-jar <jar
file>] [-inputformat <class>] [-map <class>] [-partitioner <class>]
```

```
[-reduce <class>] [-writer <class>] [-program <executable>] [-
reduces <num>]
```

COMMAND_OPTION	Description
-conf <path>	**Configuration for job**
-jobconf <key=value>, <key=value>, ...	**Add/override configuration for job**
-input <path>	**Input directory**
-output <path>	**Output directory**
-jar <jar file>	**Jar filename**
-inputformat <class>	**InputFormat class**
-map <class>	**Java Map class**
-partitioner <class>	**Java Partitioner**
-reduce <class>	**Java Reduce class**
-writer <class>	**Java RecordWriter**
-program <executable>	**Executable URI**
-reduces <num>	**Number of reduces**

queue

command to interact and view Job Queue information

```
Usage : hadoop queue [-list] | [-info <job-queue-name> [-showJobs]]
```

COMMAND_OPTION	Description
`-list`	**Gets list of Job Queues configured in the system. Along with scheduling information associated with the job queues.**
`-info <job-queue-name> [-showJobs]`	**Displays the job queue information and associated scheduling information of particular job queue. If -showJobs options is present a list of jobs submitted to the particular job queue is displayed.**

version

Prints the version.

```
Usage: hadoop version
```

CLASSNAME

hadoop script can be used to invoke any class.

```
Usage: hadoop CLASSNAME
```

Runs the class named CLASSNAME.

Administration Commands

Commands useful for administrators of a hadoop cluster.

balancer

Runs a cluster balancing utility. An administrator can simply press Ctrl-C to stop the rebalancing process. See Rebalancer for more details.

```
Usage: hadoop balancer [-threshold <threshold>]
```

COMMAND_OPTION	Description
`-threshold`	**Percentage of disk capacity. This overwrites**

<threshold>	**the default threshold.**

daemonlog

Get/Set the log level for each daemon.

```
Usage: hadoop daemonlog -getlevel <host:port> <name>
Usage: hadoop daemonlog -setlevel <host:port> <name> <level>
```

COMMAND_OPTION	Description
-getlevel <host:port> <name>	**Prints the log level of the daemon running at <host:port>. This command internally connects to http://<host:port>/logLevel?log=<name>**
-setlevel <host:port> <name> <level>	**Sets the log level of the daemon running at <host:port>. This command internally connects to http://<host:port>/logLevel?log=<name>**

datanode

Runs a HDFS datanode.

```
Usage: hadoop datanode [-rollback]
```

COMMAND_OPTION	Description
-rollback	**Rollsback the datanode to the previous version. This should be used after stopping the datanode and distributing the old hadoop version.**

dfsadmin

Runs a HDFS dfsadmin client.

`Usage: hadoop dfsadmin [GENERIC_OPTIONS] [-report] [-safemode enter | leave | get | wait] [-refreshNodes] [-finalizeUpgrade] [-upgradeProgress status | details | force] [-metasave filename] [-setQuota <quota> <dirname>...<dirname>] [-clrQuota <dirname>...<dirname>] [-help [cmd]]`

COMMAND_OPTION	Description
`-report`	**Reports basic filesystem information and statistics.**
`-safemode enter | leave | get | wait`	**Safe mode maintenance command. Safe mode is a Namenode state in which it 1. does not accept changes to the name space (read-only) 2. does not replicate or delete blocks. Safe mode is entered automatically at Namenode startup, and leaves safe mode automatically when the configured minimum percentage of blocks satisfies the minimum replication condition. Safe mode can also be entered manually, but then it can only be turned off manually as well.**
`-refreshNodes`	**Re-read the hosts and exclude files to update the set of Datanodes that are allowed to connect to the Namenode and those that should be decommissioned or recommissioned.**

-finalizeUpgrade	**Finalize upgrade of HDFS. Datanodes delete their previous version working directories, followed by Namenode doing the same. This completes the upgrade process.**
-upgradeProgress status | details | force	**Request current distributed upgrade status, a detailed status or force the upgrade to proceed.**
-metasave filename	**Save Namenode's primary data structures to <filename> in the directory specified by hadoop.log.dir property. <filename> will contain one line for each of the following** **1. Datanodes heart beating with Namenode** **2. Blocks waiting to be replicated** **3. Blocks currrently being replicated** **4. Blocks waiting to be deleted**
-setQuota <quota> <dirname>...<dirname>	**Set the quota <quota> for each directory <dirname>. The directory quota is a long integer that puts a hard limit on the number of names in the directory tree.** **Best effort for the directory, with faults reported if** **1. N is not a positive integer, or** **2. user is not an administrator, or** **3. the directory does not exist or is a file, or**

	4. the directory would immediately exceed the new quota.
`-clrQuota <dirname>...<dirname>`	**Clear the quota for each directory <dirname>.** **Best effort for the directory. with fault reported if** **1. the directory does not exist or is a file, or** **2. user is not an administrator.** **It does not fault if the directory has no quota.**
`-help [cmd]`	**Displays help for the given command or all commands if none is specified.**

jobtracker

Runs the MapReduce job Tracker node.

```
Usage: hadoop jobtracker
```

namenode

Runs the namenode. More info about the upgrade, rollback and finalize is at Upgrade Rollback

```
Usage: hadoop namenode [-format] | [-upgrade] | [-rollback] | [-finalize] | [-importCheckpoint]
```

COMMAND_OPTION	Description
`-format`	**Formats the namenode. It starts the namenode, formats it and then shut it down.**
`-upgrade`	**Namenode should be started with upgrade option after the distribution of new hadoop**

	version.
`-rollback`	**Rollsback the namenode to the previous version. This should be used after stopping the cluster and distributing the old hadoop version.**
`-finalize`	**Finalize will remove the previous state of the files system. Recent upgrade will become permanent. Rollback option will not be available anymore. After finalization it shuts the namenode down.**
`-importCheckpoint`	**Loads image from a checkpoint directory and save it into the current one. Checkpoint dir is read from property fs.checkpoint.dir**

secondarynamenode

Runs the HDFS secondary namenode. See Secondary Namenode for more info.

```
Usage: hadoop secondarynamenode [-checkpoint [force]] | [-geteditsize]
```

COMMAND_OPTION	Description
`-checkpoint [force]`	**Checkpoints the Secondary namenode if EditLog size >= fs.checkpoint.size. If -force is used, checkpoint irrespective of EditLog size.**
`-geteditsize`	**Prints the EditLog size.**

tasktracker

Runs a MapReduce task Tracker node.

```
Usage: hadoop tasktracker
```