< HADOOP MAP_REDUCE>

Big Data - 1η ομαδική εργασία

<Γιώργος Παπαστεργιόπουλος, dai17037>

<Παναγιώτης Σκοπιανός, it14196>

07/12/2022

1. Εισαγωγή

Το παρόν έγγραφο αποτελεί μια αναφορά για την επίλυση του προβλήματος καταμέτρησης συχνών συναλλαγών με **mapReduce**. Ακόμη, παρουσιάζεται ο αλγόριθμος, οι χρόνοι που καταμετρήθηκαν καθώς παρατηρήσεις και συμπεράσματα.

Τα πειράματα έγιναν για α) έναν κόμβο, β) δύο κόμβους και με κατώφλια τις παρακάτω τιμές συναλλαγών: 5000, 10000, 50000. Για κάθε μία από τις παραπάνω διαμορφώσεις, πραγματοποιήσαμε 5 εκτελέσεις και αφού αφαιρέσαμε τη μεγαλύτερη και μικρότερη τιμή, καταγράψαμε το μέσο όρο των 3 υπολοίπων εκτελέσεων.
Οι εκτελέσεις των πειραμάτων έγιναν με την χρήση 4 reducers.

Στον φάκελο υπάρχουν:

- ένας φάκελος **output** που περιέχει τα αρχεία εξόδου των εκτελέσεων
- ένας φάκελος **TransactionsCount** που περιέχει το κώδικα για ένα job.
- ένας φάκελος **TransactionsCountChain** που περιέχει το κώδικα για δύο jobs.
- ένα αρχείο **execution_times.xlsx** με καταγεγραμμένους όλους τους χρόνους εκτέλεσης
- τα **jar** αρχεία και για τα δύο προγράμματα

Για την εκτέλεση του TransactionsCount

~/hadoop/bin/hadoop jar ~/hadoop/share/hadoop/mapreduce/ TransactionsCount-0.0.1.jar grep input output threshold reducers

Όπου:

1. **input** = το αρχείο εισόδου
2. **output** = το αρχείο εξόδου
3. **threshold** = το όριο του αριθμού εμφάνισης (5000,10000,...)
4. **reducers** = ο αριθμός των reducers(1,2,4,…)

Για την εκτέλεση του TransactionsCountChain

~/hadoop/bin/hadoop jar ~/hadoop/share/hadoop/mapreduce/ TransactionsCountChain-0.0.1.jar grep input temp output threshold reducers

Όπου:

1. **input** = το αρχείο εισόδου
2. **temp** = το αρχείο εξόδου του πρώτου **job** δίνεται σαν είσοδος στο δεύτερο job
3. **output** = το αρχείο εξόδου
4. **threshold** = το όριο του αριθμού εμφάνισης (5000,10000,...)
5. **reducers** = ο αριθμός των reducers(1,2,4,…)

2. Αλγόριθμος

Αρχικά για την επίλυση του προβλήματος με **MapReduce** σε κατανεμημένη διαμόρφωση ο αλγόριθμος λειτουργεί ως εξής:

Οι **mappers** χωρίζουν τις γραμμές από το αρχείο εισόδου σε tokens που το κάθε ένα αντιστοιχεί σε κάθε συναλλαγή.Για κάθε ένα token θα υπολογιστούν όλα τα υποσύνολα προϊόντων(δηλαδή όλους τους συνδυασμούς προϊόντων σε μια συναλλαγή) που υπάρχουν.Στη συνέχεια τα ταξινομεί και τα γράφει στο δίσκο.

Οι **combiners** αναλαμβάνουν μια αθροιστική λειτουργία(παρόμοια με αυτή των **reducers**).Ο ρόλος τους είναι να προσθέτει τις **τιμές(values)** των καταχωρήσεων με ίδιο **κλειδί(key)**.Η απόφαση για το αν θα κληθούν ή όχι εξαρτάται από το Hadoop καθώς η λειτουργία τους είναι βοηθητική στο να μειώσει την συμφόρηση στο δίκτυο όταν αποστέλλονται τα δεδομένα από τους **mappers** στους **reducers**.

Οι **reducers** έχουν την ίδια λειτουργία με τους **combiners** εκτός από ότι ελέγχουν αν το **sum(value)** είναι μεγαλύτερο ή ίσο από το όριο που έχει εισάγει ο χρήστης.Εκτελούνται πάντα, όταν όμως έχουν τελειώσει την εκτέλεση τους όλοι οι **mappers.**

Η επίλυση του προβλήματος όμως έγινε με την χρήση δύο διαφορετικών **δουλειών(jobs)** καθώς θα μπορούσαμε να περιορίσουμε τον αριθμό των πράξεων κατά τον υπολογισμό των συνδυασμών.

Πρώτη φάση mapReduce(job1)

- Οι **mappers** χωρίζουν το αρχείο σε γραμμές και παράγουν ένα αρχείο της μορφής ***<Συναλλαγή_n, 1>***.
- Οι **combiners** θα προσθέσουν τις τιμές των ίδιων κλειδιών.Οπότε το αρχείο εξόδου θα είναι της μορφής ***<Συναλλαγή_1, 20>,<Συναλλαγή_2, 2>.....***
- Οι **reducers** επαναλαμβάνουν την δουλεία των combiners χωρίς όμως να εκτελούν περαιτέρω ελέγχους.

Δεύτερη φάση mapReduce(job2)

- Οι **mappers** τώρα παίρνουν ως είσοδο ένα αρχείο που περιέχει σε κάθε γραμμή μια συναλλαγή και τον αριθμό του πόσες φορές έχει εμφανιστεί.Θα υπολογίσει για κάθε γραμμή τους συνδυασμούς των στοιχείων,θα τους ταξινομήσει βάση του εσωτερικού κλειδιού και θα τους γράψει ως εξής: ***<Συνδιασμός_n, αριθμός εμφάνισης>***.
- Οι **combiners** θα προσθέσουν τις τιμές των ίδιων κλειδιών.Οπότε το αρχείο εξόδου θα είναι της μορφής ***<Συνδυασμός_1, 20>,<Συνδιασμός_2, 7>.....***
- Οι **reducers** έχουν την ίδια λειτουργία με τους **combiners** εκτός από ότι ελέγχουν αν το **sum(value)** είναι μεγαλύτερο ή ίσο από όριο που έχει εισάγει ο χρήστης.

3. Περιορισμοί και θεωρήσεις

1. Αρχικά για να περάσουμε την παράμετρο του **threshold** που εισάγουμε κατά την εκτέλεσης της εντολής και για να μπορέσει να έχει πρόσβαση σε αυτήν ο **reducer** στο **Configuration file** που δημιουργήσαμε εισάγαμε για κλειδί το **'threshold'** με τιμή το τέταρτο όρισμα(**args[4]**).Αντίστοιχα στον **reducer** μπορούμε να αποκτήσουμε πρόσβαση σε αυτό όπως φαίνεται παρακάτω.

```
String trans = context.getConfiguration().get("threshold");
```

2. Στο **job1** επειδή και ο **reducer** και ο **combiner** εκτελούν ακριβώς την ίδια λειτουργία έχουμε ορίσει μόνο την **Combiner.class** και στους δύο.

```
Configuration conf = new Configuration();
conf.set("threshold", args[4]);

// JOB 1
Job job1 = Job.getInstance(conf, jobName: "job1-transactionsCount_2nodes_N=50000");
job1.setJarByClass(TransactionsCount.class);
job1.setMapperClass(TokenizerMapper1.class);
job1.setCombinerClass(Combiner.class);
job1.setReducerClass(Combiner.class);
job1.setNumReduceTasks(Integer.parseInt(args[5]));
job1.setOutputKeyClass(Text.class);
job1.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job1, new Path(args[1]));
FileOutputFormat.setOutputPath(job1, new Path(args[2]));
job1.waitForCompletion( verbose: true);

// JOB 2
Job job2 = Job.getInstance(conf, jobName: "job2-transactionsCount_2nodes_N=50000");
job2.setJarByClass(TransactionsCount.class);
job2.setMapperClass(TokenizerMapper2.class);
job2.setCombinerClass(Combiner.class);
job2.setReducerClass(IntSumReducer.class);
job2.setNumReduceTasks(Integer.parseInt(args[5]));
job2.setOutputKeyClass(Text.class);
job2.setOutputValueClass(IntWritable.class);

FileInputFormat.addInputPath(job2, new Path(args[2]));
FileOutputFormat.setOutputPath(job2, new Path(args[3]));
return job2.waitForCompletion( verbose: true) ? 0 : 1;
```

Κώδικας για το στήσιμο των jobs.

Καθώς το αποτέλεσμα από το **job1** τροφοδοτείται στην είσοδο του **job2**,το δεύτερο παίρνει ως ορίσματα το **args[2]** για το αρχείο εισόδου και το **args[3]** για το αρχείο εξόδου.Το **job1** θα πρέπει να ολοκληρωθεί πρώτα για να ξεκινήσει το **job2**.

3. Στο **job2** ο **mapper** παίρνει σαν είσοδο ένα αρχείο που περιέχει τις συναλλαγές σε αυτήν την μορφή:

   <Συναλλαγή_1, αριθμός εμφανίσεων_1>
   <Συναλλαγή_2, αριθμός εμφανίσεων_2>
   <Συναλλαγή_n, αριθμός εμφανίσεων_n>

   αφού τα χωρίσει σε **tokens** και τα βάλει σε μια λίστα με την μορφή:

   [προιόν_1,προιόν_2,...,προιόν_n,αριθμός εμφάνισης_1]

   αφαιρεί το τελευταίο στοιχείο αυτής που είναι ο αριθμός εμφάνισης και το θέτει στην μεταβλητή για την **τιμή** του κάθε **κλειδιού**.

```
public static class TokenizerMapper2 extends Mapper<Object, Text, Text, IntWritable> {
    2 usages
    private static IntWritable count = new IntWritable();
    2 usages
    private Text word = new Text();

    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        StringTokenizer lineToken = new StringTokenizer(value.toString(), delim: ",[]\t ", returnDelims: false);
        ArrayList<String> lineArray = new ArrayList<String>();
        while (lineToken.hasMoreElements()) {
            lineArray.add((lineToken.nextToken()));
        }
        int lastIndex = lineArray.size() - 1;
        count.set(Integer.parseInt(lineArray.get(lastIndex)));
        lineArray.remove(lastIndex);
```

4. Κάποιοι από τους παραγόμενους συνδυασμούς για παράδειγμα οι συνδυασμοί (F1,F2) και (F2,F1) δεν φαίνεται ότι είναι οι ίδιοι, παρόλου που είναι.Για να αποφύγουμε τέτοιες περιπτώσεις μπορούμε να ταξινομήσουμε τις συναλλαγές ώστε οι παραγόμενοι συνδυασμοί να εμφανίζονται με ταξινομημένη σειρά.

5. Επίσης παρατηρήθηκε στις αρχικές εκτελέσεις χωρίς την χρήση των **combiners,** υπήρξε μια σημαντική επιβράδυνση όταν τελείωναν οι **mappers** την εκτελεσή τους και έπρεπε να στείλουν τα δεδομένα στους **reducers**.Καθώς αντί να αποστέλλονται N εγγραφές του τύπου <key, 1>,μετά από τον **combiner** αποστέλλονται <key, N>,όπου N το πλήθος των ίδιων κλειδιών(key).

6. Η επίλυση του προβλήματος προτιμήθηκε να γίνει με **mapReduce chaining jobs** γιατί θεωρήσαμε πως μπορούμε να μειώσουμε το αριθμό των υπολογισμών αν ‘σπάσουμε’ το αρχικό task σε δύο.Αντί ο **mapper** να βρίσκει τους συνδυασμούς για κάθε συναλλαγή στο αρχείο,θα μπορούσαμε να έχουμε 2 tasks που στο πρώτο task βρισκουμε και προσθέτουμε μόνο τις συναλλαγές που επί της ουσίας ομαδοποιούμε ίδιες συναλλαγες και στο δεύτερο task για κάθε μια από τις συναλλαγές υπολογίζουμε μια φορά τους συνδιασμούς και σαν τιμή βάζουμε το πλήθος εμφάνισης τους.Έτσι για παράδειγμα αν η συναλλαγή (F1,F2)(1) εμφανίζεται στο αρχείο 1000 φορές δεν χρειάζεται να υπολογίσουμε 1000 φορές τους συνδυασμούς της (1).Αρκεί μόνο μια φορά και απλά σαν τιμή(value) καταχωρούμε το 1000.

Παράδειγμα εκτέλεσης

Αρχείο εισόδου	*Έξοδος πρώτου job*
F1 F5 F2 F10	[F1, F10, F2, F5] 1
F10 F3 F1	[F1, F10, F3] 1
F2 F3	[F1, F2, F4, F5] 1
F5	[F2, F3] 1
F4 F5 F1 F2	[F5] 1

Έξοδος δεύτερου job

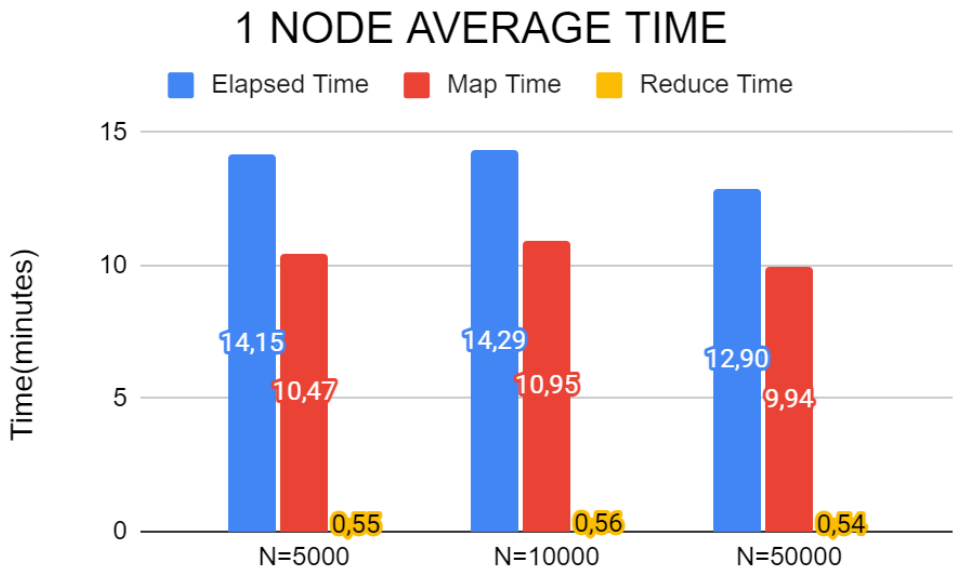
[F1, F10, F2, F5] 1	[F2, F4, F5] 1
[F1, F10, F2] 1	[F2, F4] 1
[F1, F10, F3] 1	[F2, F5] 2
[F1, F10, F5] 1	[F2, F5] 2
[F1, F10] 2	[F2] 3
[F1, F2, F4, F5] 1	[F3] 2
[F1, F2, F4] 1	[F4, F5] 1
[F1, F2, F5] 2	[F4] 1
[F1, F2] 2	[F1, F3] 1
[F5] 3	
[F1, F4, F5] 1	
[F1, F4] 1	
[F1, F5] 2	
[F10, F2, F5] 1	
[F10, F2] 1	
[F10, F3] 1	
[F10, F5] 1	
[F10] 2	
[F1] 3	
[F2, F3] 1	

4. Μετρήσεις

Παρακάτω παρουσιάζονται οι χρόνοι εκτέλεσης για το TransactionsCountChain-0.0.1.jar

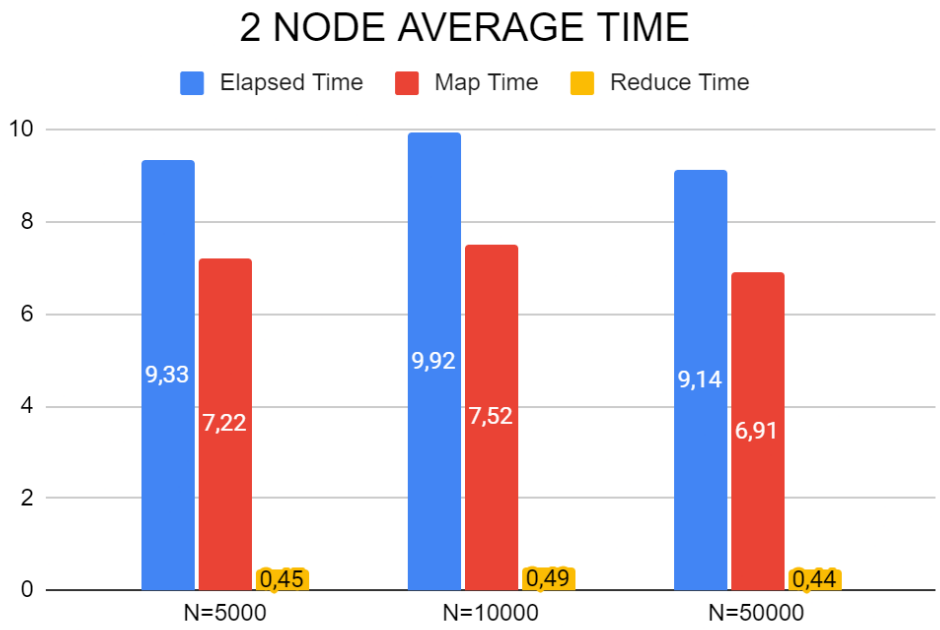
Χρόνοι εκτέλεσης για 1 κόμβο

	1 NODE AVERAGE TIME		
	Elapsed Time	**Map Time**	**Reduce Time**
N=5000	14,15	10,47	0,55
N=10000	14,29	10,95	0,56
N=50000	12,90	9,94	0,54



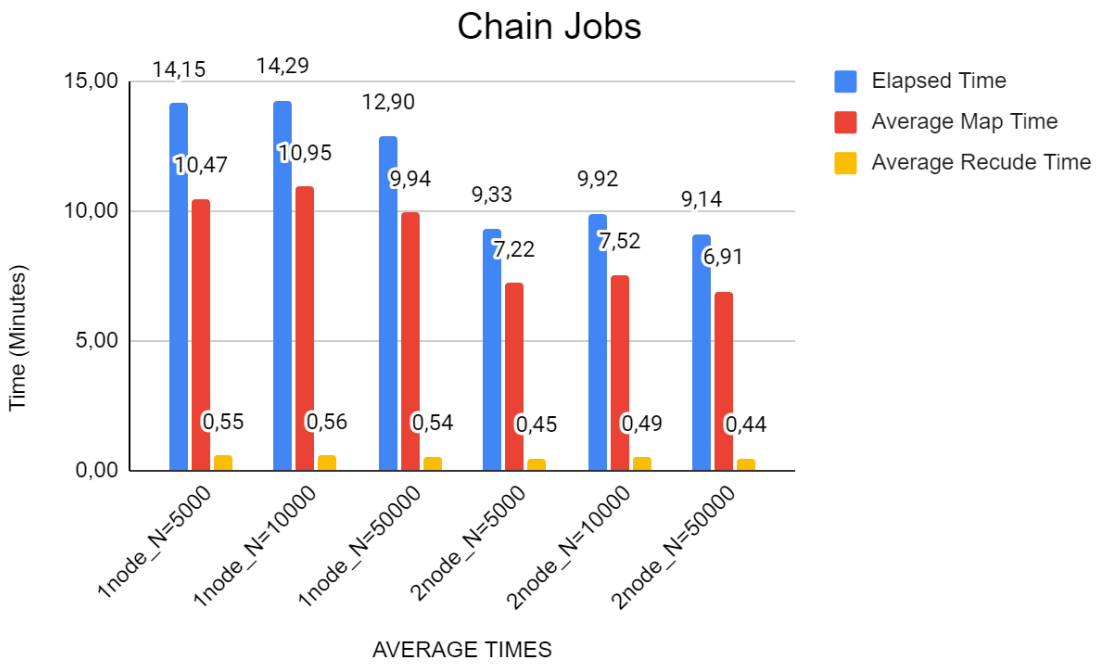
Χρόνοι εκτέλεσης για 2 κόμβους

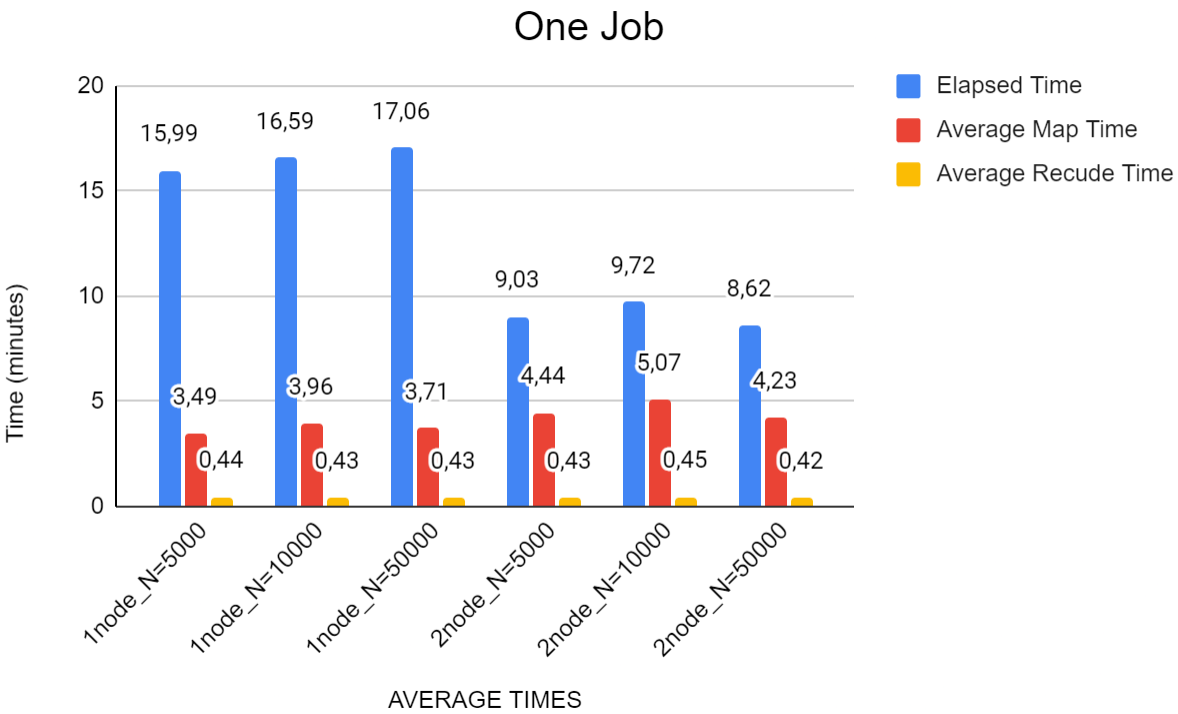
	2 NODE AVERAGE TIME		
	Elapsed Time	Map Time	Reduce Time
N=5000	9,33	7,22	0,45
N=10000	9,92	7,52	0,49
N=50000	9,14	6,91	0,44



Αρχικά παρατηρούμε ότι υπάρχει σημαντική βελτίωση στους χρόνους(**~30%**) όταν χρησιμοποιούμε **2-nodes** στο **cluster** μας.Ακόμα και στις δύο εκτελέσεις για τις διάφορες τιμές του N παρουσιάζουν παρόμοια συμπεριφορά στους χρόνους.Καθώς το N μεγαλώνει ο χρόνος των **reducers** και **mappers** μειώνεται.Εξαίρεση αποτελεί όταν το N=10000,που θα ήταν αναμενόμενο να υπάρχει μια μείωση του χρόνου,ενώ αντιθέτως παρατηρούνται και οι χειρότεροι χρόνοι εκτελεσης.

Σύγκριση εκτέλεσης Chain Jobs - Single Job





Η διαφορά των **mapper** στο **OneJob** είτε με ένα **node** είτε με δύο **node** είναι αμελητέα.Ενώ στο **ChainJobs** η διαφορά είναι σημαντική γιατί πραγματοποιούμε δύο φορές **I/O** στον δίσκο.Αυτό φαίνεται και στην αναμεταξύ τους σύγκριση στους χρόνους των mappers για τον ίδιο λόγο με το παραπάνω.
Ακόμα για ένα **node** στο **OneJob** καθώς αυξάνεται το N μεγαλώνει και ο συνολικός χρόνος(**Elapsed Time**) σε αντίθεση με το **ChainJobs**, ενώ στην εκτέλεση για 2 **nodes** και τα δύο παρουσιάζουν παρόμοιες συμπεριφορές.
Ένα τελευταίο σημείο που αξίζει να σημειωθεί είναι ότι στο **ChainJobs** καταναλώνεται πολύ περισσότερος χρόνος στο κομμάτι του mapping και λιγότερος στο κομμάτι **Shuffle & Merge**,σε αντίθεση με το **OneJob**.

Το **ChainJob** φαίνεται να τα πηγαίνει πολύ καλύτερα με ένα **node** γιατί τα **I/O** στο δίσκο και από τα δύο **jobs** δεν φαίνεται να επηρεάζουν τόσο το χρόνο, σε αντίθεση με το χρόνο που χρειάζεται να υπολογίσει όλους τους συνδυασμούς όσες φορές και να υπάρχουν.
Με δύο **nodes** γρηγορότερος είναι ο **OneJob**.