Comparison of Distributed Computing Approaches to Complexity of *n*-gram Extraction

Sanzhar Aubakirov[1], Paulo Trigo[2] and Darhan Ahmed-Zaki[1]
[1]*Department of Computer Science, al-Farabi Kazakh National University, Almaty, Kazakhstan*
[2]*Instituto Superior de Engenharia de Lisboa, Biosystems and Integrative Sciences Institute / Agent and Systems Modeling, Lisbon, Portugal*

Keywords: Distributed Computing, Text Processing, *n*-gram Extraction.

Abstract: In this paper we compare different technologies that support distributed computing as a means to address complex tasks. We address the task of *n*-gram text extraction which is a big computational given a large amount of textual data to process. In order to deal with such complexity we have to adopt and implement parallelization patterns. Nowadays there are several patterns, platforms and even languages that can be used for the parallelization task. We implemented this task on three platforms: (1) MPJ Express, (2) Apache Hadoop, and (3) Apache Spark. The experiments were implemented using two kinds of datasets composed by: (A) a large number of small files, and (B) a small number of large files. Each experiment uses both datasets and the experiment repeats for a set of different file sizes. We compared performance and efficiency among MPJ Express, Apache Hadoop and Apache Spark. As a final result we are able to provide guidelines for choosing the platform that is best suited for each kind of data set regarding its overall size and granularity of the input data.

1 INTRODUCTION

Applications of many algorithms and methods of text analysis depends on statistical information about text (Riedl and Biemann, 2012), statistics about *n*-grams are an important building block in knowledge discovery and information retrieval (Berberich and Bedathur, 2013). In this paper we compare different approaches that deal with *n*-gram extraction tasks via the distribution of computation and data across clusters of processing and storage resources. The evaluated approaches follow a model where the "computation moves to the data" (instead of the data being transferred to feed the computation). We follow this model, of moving code to available resources, to deal with the *n*-gram extraction task where the temporal and space complexity grows with the number, *n*, of grams to extract. We adopted the current patterns of task parallelization, such as the map-reduce paradigm, and implemented the *n*-gram extraction task using three different platforms, namely the MPJ Express, the Apache Hadoop and the Apache Spark. We also describe application architecture developed for MPJ Express implementation in order to provide reliability and fault-tolerance. The contributions from our work include:

- comprehensive experimental evaluation on English Wikipedia articles corpora
- time and space comparison between implementations on MPJ Express, Apache Hadoop and Apache Spark
- detailed guidelines for choosing platform regarding size and granularity of the input data

We will start from introduction to each platform and assumptions made, then we will introduce *n*-gram extraction method and show results of the experiments. Finally we will provide pros and cons of each platform regarding particular data set.

1.1 Apache Hadoop

Apache Hadoop is a Java Virtual Machine (JVM) based framework that implements Map/Reduce paradigm. It is dividing main task into many small fragments of work, each of which may be executed or re-executed on any node in the cluster. Map/Reduce paradigm is very well suited for the text processing tasks because input data could be divided into equal chunks and processed separately (Lin and Dyer,

Aubakirov, S., Trigo, P. and Ahmed-Zaki, D.
Comparison of Distributed Computing Approaches to Complexity of *n*-gram Extraction.
DOI: 10.5220/0005943000250030
In *Proceedings of the 5th International Conference on Data Management Technologies and Applications (DATA 2016)*, pages 25-30
ISBN: 978-989-758-193-9

2010). One of the main features of Apache Hadoop is transparently provided reliability, fault tolerance and data motion. It sends same task to a different nodes that provides ability to avoid downtime in case of failure of one of the computing nodes, but at the same time it decrease overall performance of the cluster. Hadoop Distributed File System (HDFS) used for nodes communication and as temporary data storage. HDFS abstraction layer provides ability to process big amount of data that do not fits into fast random-access memory. At the same time it is reducing performance, seek times for random disk access are fundamentally limited by the mechanical nature of the devices.

1.2 Apache Spark

Apache Spark is a JVM based framework that uses abstraction layer for cluster communication named Resilient Distributed Dataset (RDD). RDD provides ability to store data as a certain collections distributed in the cluster memory (Zaharia et al., 2012). Apache Spark uses micro-batch task execution model based on techonology called D-Streams. D-Streams technology streaming computation as a series of stateless and deterministic batch computations on small time intervals (Zaharia et al., 2013). Both RDD and D-Streams provide reliability and fault-tolerance of task execution, because the system can recover all intermediate state and results on failure. Also Apache Spark can be easily integrated with HDFS and Apache Yarn.

1.3 MPJ Express

MPJ Express is Java implementation of Message Passing Interface (MPI). The MPI has become a de facto standard for writing High Performance Computing (HPC) applications on clusters and Massively Parallel Processors (MPP). It is inherits all advantages and disadvantages of MPI. The main disadvantages is the difficulty of writing and debugging distributed applications. The main advantages are flexibility, customizability and high performance of the application.

MPJ do not have built-in task manager that provides fault-tolerance and reliability, thus failure of one of executors leads to task failure. We propose application architecture in order to compete with applications based on Apache Spark and Apache Hadoop, it is shown on the figure 1.

In order to minimize communication between nodes and reduces network latency overhead each cluster node processing one file at a time. We are using HDFS to avoid data motion and transactional queue synchronizing pattern to provide reliability.

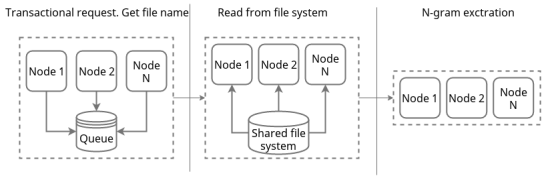

Figure 1: Architecture of application based on MPJ Express.

2 METHOD

We are extracting n-gram statistical model from the Wikipedia articles corpora. The data set description is shown in table 1. The overall corpora size is 4 Gb and consist of 209716 articles, each article's size is approximately 20 Kb. In order to cover most possible data set types we divided all corpora into 6 volumes: 64 Mb, 256 Mb, 512 Mb, 1024 Mb, 2048 Mb and 4096 Mb. Each volume is divided into two sets: A) a large number of small files, and B) a small number of large files. For data set A we keep Wikipedia articles as is, for B data set we concatenate articles into bigger files.

Table 1: Description of the corpora.

size	# articles	# tokens
64 Mb	3277	$6 * 10^6$
256 Mb	13108	$25 * 10^6$
512 Mb	26215	$51 * 10^6$
1024 Mb	52428	$102 * 10^6$
2048 Mb	104858	$206 * 10^6$
4096 Mb	209716	$412 * 10^6$

Our goal is to extract statistical n-gram model from all corpora and from each article separately. We consider that full n-gram model is the all extracted n-grams, where $n \in [1,k]$ and k is the length of longest sentence in the dataset. We are using method that is described by Google in their paper (Brants et al., 2007) and consider improvements suggested by work (Berberich and Bedathur, 2013). Both algorithms based on MapReduce paradigm. Method proposed by (Berberich and Bedathur, 2013) optimized memory consumption overall performance, but at the same time rejecting not frequent n-grams. For experiments we are using Google's algorithm because our goal is to obtain full n-gram model.

We adopted algorithm for our goal of n-gram extraction from the individual articles. Our method operates with sentences, text of the articles is represented as set of sentences S, where $S = (S_1, S_2, S_3, \ldots, S_n)$, and each sentence S_n is a list of

words $S_n = (W_1, W_2, W_3, \ldots, W_m)$, where W_n is a single word.

Algorithm 1: Pseudo code for n-gram extraction.

```
1: function MAP(list sentence, int size)                    ▷ map
2:     return rangeClosed(1, size)
3:     .map(n → sliding(sentence, n, size))
4: end function
5: function REDUCE(stream ngrams)                           ▷ reduce
6:     return ngrams.collect(groupByCount())
7: end function
8: function SLIDING(list sentence, int n, int size)
9:     return rangeClosed(0, size - n)
10:    .map(idx → join(" ", list.subList(idx, idx + n)))
11: end function
```

We implemented *sliding*(), *map*() and *reduce*() functions, pseudo-code is shown in figure 1. Function *map*() takes list of sentences S and for each S_i executes *sliding*() function with the parameter $n = (0, 1, 2, \ldots, m)$, where n is size of slides (n-grams) that function will produce and m is number of words W in sentence S_i. Function *reduce*() takes output of *map*() function, which is the list of n-grams (list of list of words) and count similar ones. As a results it returns list of objects $(n\text{-}gram, v)$, that is usually called Map, where v is the frequency of particular n-gram in the text. This approach provide ability to execute independent *map*() and avoid communication between nodes until *reduce*() stage.

2.1 MPJ Express Implementation

We are using transactional queue synchronization pattern as a process task management for MPJ Express. Transactions guarantees message delivery to a single recipient. Thus in case of failure transaction can be aborted and message will be delivered to another worker. We fill queue with the file names, each node takes file name from queue, process it and takes another one. This solution provides reliability, fault-tolerance and scalability. And one more benefit is that workers can be implemented in any language or platform, not only JVM based applications.

2.2 Apache Hadoop Implementation

Word count application is probably the most common example of Apache Hadoop usage. We use algorithm from the Apache Hadoop community official site as basis for our task. The count part remaining almost the same, with the adoptions to n-gram extraction task. Pseudo-code of *map*() and *reduce*() functions shown on Algorithm 2.

Algorithm 2: Pseudo-code of Apache Hadoop implementation.

```
1: function MAP(key, value, context)                        ▷ map
2:     list sentences = getSentences(value)
3:     for (sentence : sentences) do
4:         for (n← 1, sentence.size()) do
5:             ngrams = sliding(sentence, n)
6:             ngrams
7:                 .forEach(ngram → write(ngram, 1))
8:         end for
9:     end for
10: end function
11: function REDUCE(key, values, context)                   ▷ reduce
12:     sum = 0
13:     for (value : values) do
14:         sum += value.get()
15:     end for
16:     result.set(sum)
17:     write(key, result)
18: end function
```

2.3 Apache Spark Implementation

Applications based on Apache Spark framework can be implemented using one of three programming languages: Python, Scala and Java. Scala is a functional programming language that fully executes in JVM. It provide advantages such as immutable data structures, type safety and pure functions. This language features simplifies code parallelization by reducing the biggest headache in distributed programming such as race conditions, deadlocks, and other well-known problems. As a results code written in Scala become cleaner, shorter and easy to read. Apache Spark application was implemented in Scala, pseudo-code of implementation is described in Algorithm 3.

Algorithm 3: Pseudo-code of Apache Spark map and reduce functions.

```
1: function MAP(sentence, N)                                ▷ map
2:     (1 to N).toStream.flatMap(n ⇒ sentence.sliding(n))
3: end function
4: function REDUCE(text)                                    ▷ reduce
5:     text.flatMap(sentence ⇒ sliding(sentence))
6:         .map(ngram ⇒ (ngram,1))
7:         .reduceByKey((a,b) ⇒ a+b)
8:         .saveAsTextFile()
9: end function
```

2.4 Assumptions

We implemented n-gram extraction library that is used by Hadoop and MPJ, while Spark use built-in functions. All implementations use HDFS for

input/output tasks and there is one separate machine that serves it. MPJ work with files splitted in chunks, while both Spark and Hadoop works with data streams.

Spark and Hadoop use built-in task management, while MPJ implementation use messaging queue. Messaging queue service was installed on separate server and was not included into a cluster as a usual node.

3 EXPERIMENTAL RESULTS

Technical characteristics of the cluster is shown in tables 2 and 3. There are 16 nodes, each node has the same characteristics. Figure 2 shows overall picture for results of the experiments, parallelization gives good efficiency and speedup on all platforms.

Table 2: Cluster specification.

CPU	RAM	HDD	Net
Intel Core i5-2500 3.30GHz	16Gb	500GB 7200RPM 6Gb/s	1Gbit/s

Table 3: Software specification.

Name	Version
MPJ Express	0.44
Apache Hadoop	2.6.0
Apache Spark	1.5.0
Java	1.8.0_60
Scala	2.11.7
Ubuntu OS	14.04

During our experiments Apache Hadoop shows inefficient processing time for data sets of type A (a large number of little files). Processing time of even the most smallest data set expressed in hours. Researches (Andres and A, 2013), (Vorapongkitipun and Nupairoj, 2014) and (Andrews and Binu, 2013) shows that Apache Hadoop works faster if input data is represented as few big files instead of many little files. This is because of HDFS design, which was developed for processing big data streams. Readings of many little files leads to many communications between nodes, many disk head movements and as a consequence leads to extremely inefficient work of HDFS.

Figure 3 shows results of experiments with dataset of type A. As we mentioned before Apache Hadoop is ineffective for such type of dataset, we exclude its results from the graph. For this type of data Apache Spark application processing time is higher than MPJ

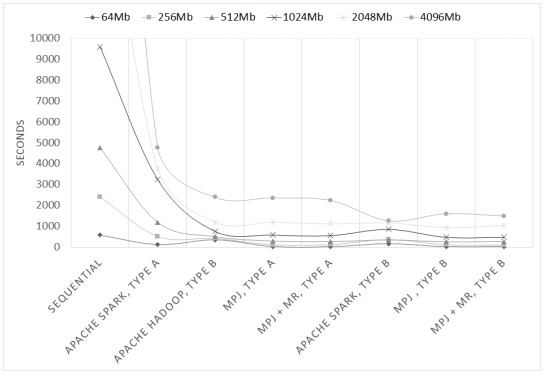

Figure 2: Shows overall picture. Simple sequential processing on the left side of the plot.

Express. The most effective time shows application based on MPJ Express platform. Both implementations MPJ and MPJ+MR shows almost the same time, but in an average MPJ+MR is 1.5 times faster. MPJ Express application architecture was designed specifically for such kind of data type where each node processing one file at a time avoiding communication between nodes.

Figure 4 shows results of experiments with dataset B. Results are varying depending on dataset volume size. In an average Apache Hadoop is the slowest platform, MPJ Express is the fastest. For data sizes from 64 Mb to 1024 Mb MPJ Express is most effective, in an average it is 3 times faster than Apache Spark and 5 times faster than Apache Hadoop. Starting from 2048 Mb dataset sizes Apache Spark shows better effectiveness, with size of 4096 Mb it is 2 times faster than other platforms.

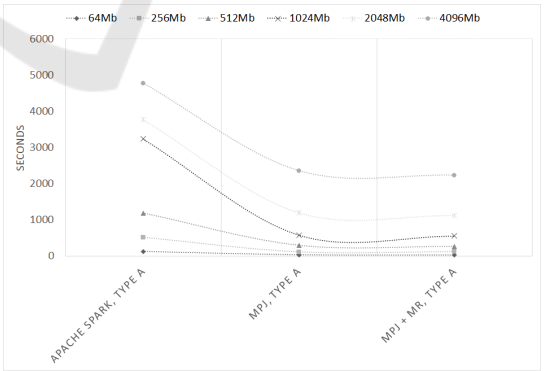

Figure 3: Experiments with dataset of type A.

4 SUMMARY

Figures 5 and 6 shows efficiency and speedup, it was computed using the best suited data set type of each platform. There is *ideal* line to simplify evaluation.

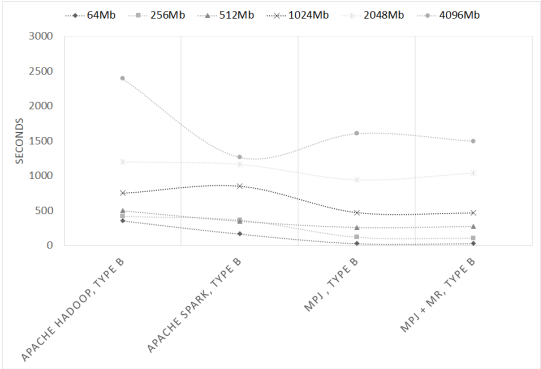

Figure 4: Experiments with dataset of type B.

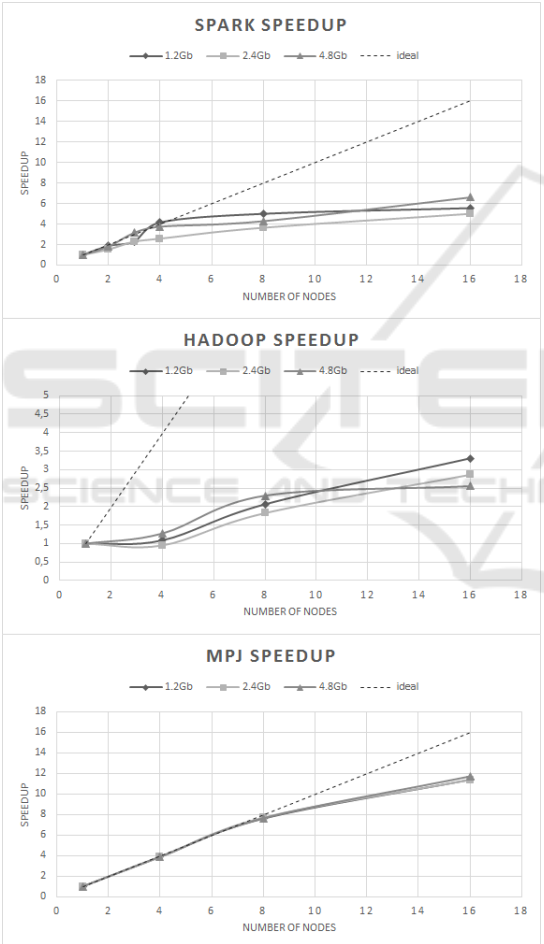

Figure 5: Shows speedup of each platform.

The graph shows that Hadoop speedup and efficiency are far from *ideal*, while MPJ is very close, Spark is always in the middle.

For the dataset type A we are dealing with a very specific task - *n*-gram model extraction for every article. It is requires to process large number of small files. Application based on MPJ Express was designed specifically for such type of data set and thus shows the highest performance. But challenges in writing distributed software using MPJ Express are great. Programmer must manage details across several threads, processes, or machines. We have to use low-level devices such as mutexes and to apply high-level design patterns such as producer-consumer queues to tackle these challenges. Concurrent programs are difficult to reason about and even harder to debug. Apache Hadoop is the platform that requires high administrative skills during cluster setup and detailed knowledge of documentation and best practices. For small data set it is considerably slower than other implementations. Apache Spark is fast and easier to configure. Code written for Apache Spark is cleaner and shorter. It is showing good efficiency on both data types and for all data set sizes. Short summary would be:

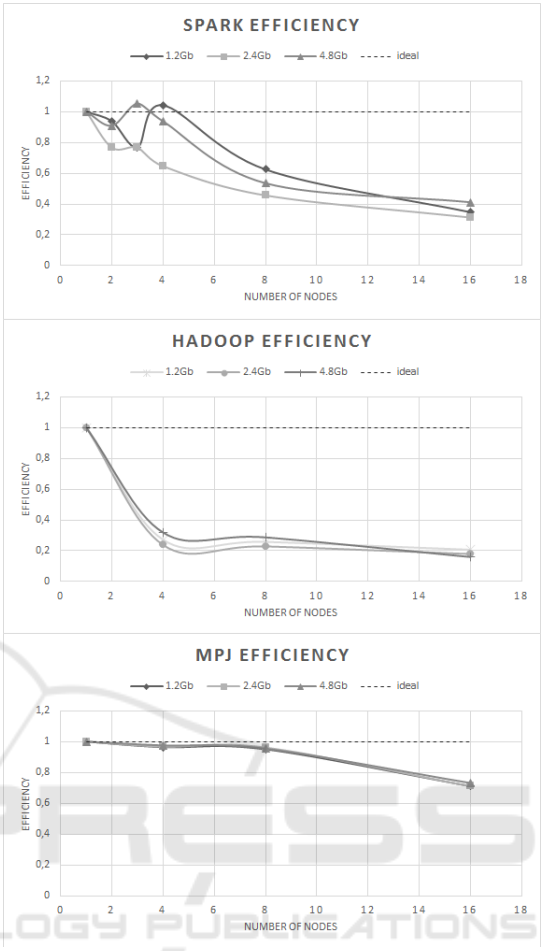

Figure 6: Shows efficiency of each platform.

- MPJ Express can be adopted for almost any task
  - Very flexible

  - The highest performance
  - Difficult to implement and to debug code
- Apache Spark is fast and easy
  - Performance is not far behind the MPJ Express
  - Clean and short code
  - Easy to configure
- Apache Hadoop
  - Slow for little dataset
  - Difficult to configure

5 CONCLUSION

We experimented on the task of extracting *n*-gram statistical model from corpora of two different types: A) a large number of small files, and B) a small number of large files. Experimental result shows that type and size of the data has a big influence on the performance of a specific platform. We have concluded that:

- for dataset type A for all data sizes MPJ Express shows the best speedup and efficiency
- for dataset type B
  - for dataset sizes of 64 Mb to 2048 Mb MPJ Express shows 3 times better speedup and efficiency
  - for dataset sizes of 2048 Mb and more Apache Spark shows 2 times better speedup and efficiency

6 COPYRIGHT FORM

The Author hereby grants to the publisher, i.e. Science and Technology Publications, (SCITEPRESS) Lda Consent to Publish and Transfer this Contribution.

REFERENCES

Andres, B. P. and A, B. (2013). Perusal on hadoop small file problem. In *Perusal on Hadoop small file problem*. IJCSEITR.

Andrews, B. P. and Binu, A. (2013). Perusal on hadoop small file problem. In *IJCSEITR*. TJPRC.

Berberich, K. and Bedathur, S. (2013). Computing n-gram statistics in mapreduce. In *EDBT '13 Proceedings of the 16th International Conference on Extending Database Technology*. EDBT.

Brants, T., Popat, A. C., Xu, P., Och, F. J., and Dean, J. (2007). Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. EMNLP-CoNLL.

Lin, J. and Dyer, C. (2010). An ontology-based approach to text summarization. In *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool.

Riedl, M. and Biemann, C. (2012). Text segmentation with topic models. In *JLCL*. JLCL.

Vorapongkitipun, C. and Nupairoj, N. (2014). Improving performance of small-file accessing in hadoop. In *JCSSE*. JCSSE.

Zaharia, M., Das, T., Li, H., Shenker, S., and Stoica, I. (2012). Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *HotCloud'12 Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing*. HotCloud.

Zaharia, M., Das, T., Li, H., Shenker, S., and Stoica, I. (2013). Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP.