Perldoop: Efficient Execution of Perl Scripts on Hadoop Clusters

José M. Abuín, Juan C. Pichel, Tomás F. Pena, Pablo Gamallo, and Marcos García
Centro de Investigación en Tecnoloxías da Información (CiTIUS)
Universidade de Santiago de Compostela
Santiago de Compostela, Spain
Email: {josemanuel.abuin,juancarlos.pichel,tf.pena,pablo.gamallo,marcos.garcia.gonzalez}@usc.es

***Abstract*—Hadoop is one of the most important implementations of the MapReduce programming model. It is written in Java and most of the programs that run on Hadoop are also written in this language. Hadoop also provides an utility to execute applications written in other languages, known as Hadoop Streaming. However, the ease of use provided by Hadoop Streaming comes at the expense of a noticeable degradation in the performance.**

In this work, we introduce Perldoop, a new tool that automatically translates Hadoop-ready Perl scripts into its Java counterparts, which can be directly executed on Hadoop while improving their performance significantly. We have tested our tool using several Natural Language Processing (NLP) modules, which consist of hundreds of regular expressions, but Perldoop could be used with any Perl code ready to be executed with Hadoop Streaming. Performance results show that Java codes generated using Perldoop execute up to 12x faster than the original Perl modules using Hadoop Streaming. In this way, the new NLP modules are able to process the whole Wikipedia in less than 2 hours using a Hadoop cluster with 64 nodes.

I. Introduction

In the modern digital society, it is estimated that each day are created around 2.5 quintillion bytes of data (2.5 Exabytes), in such a way that 90% of the data all over the world were created just only in the last two years [1]. These data come from all type of sources: sensors used to obtain information on the climate, publications in social networks, blogs, digital images and video, etc. For instance, Twitter generates about 8 Terabytes of data per day, while Facebook captures about 500 Terabytes. This is what is known as Big Data. One of the main characteristics of this amount of information is the fact that, in many cases, is not structured.

The MapReduce framework has become the de-facto standard for parallel processing of Big Data and has gained a wide adoption in both industry and research fields. One of the most successful open-source implementations based on Google's MapReduce [2] programming model is Hadoop [3], which is implemented using Java. In this model, the input and output of a MapReduce computation is a list of *(key, value)* pairs. Users only need to implement *Map* and *Reduce* functions. Each *map* produce zero or more intermediate *(key, value)* pairs by consuming one *(key, value)* pair. After this, the runtime groups automatically these intermediate *(key, value)* pairs into buckets representing *reduce* tasks. *Reduce* functions take an intermediate key and a list of values as input and produce zero or more output results.

Even though code developing in Hadoop is largely simplified with its characteristics as the automatic input splitting, task scheduling or fault tolerance mechanism, to write a Java MapReduce program is not straightforward. Besides, in some research areas, Java is not normally employed, and programmers are more familiar with other high level programming languages like Perl or Python. For example, Natural Language Processing (NLP) and Bioinformatics researchers are used to write code in Perl due to its unique ability to process text using regular expressions. These researchers have found in Hadoop Streaming the way to easily analyze big volumes (Gigabytes or even Terabytes) of textual information. However, important degradations in the performance were detected using Hadoop Streaming with respect to Hadoop Java codes [4]. Only for computational intensive jobs whose input/output size is small, the performance of Hadoop Streaming is sometimes better because of using a more efficient programming language.

For the reasons detailed above, in this paper we introduce Perldoop, a new tool that automatically translates Perl scripts prepared to be executed using Hadoop Streaming into Hadoop-ready Java codes. Our tool has been tested using several NLP modules as input, which consist of hundreds of regular expressions. In particular, three linguistic modules were considered: Name Entity Recognition (NER), PoS-Tagging and Named Entity Classification (NEC).

Performance tests were carried out on a 64 nodes Hadoop cluster. The results show that the automatically generated Java codes execute up to $12\times$ faster than the original Perl modules using Hadoop Streaming. In this way, the new NLP modules are able to process the whole Wikipedia in less than 2 hours.

II. The Perldoop tool

As it was stated in the introduction, our objective is to translate Hadoop Streaming codes written in Perl to its Java equivalents, in order to take advantage of the higher performance of Java codes in Hadoop [4]. The general case of automatically translating an arbitrary Perl code into its Java equivalent is a very hard problem, due to the characteristics of both languages. One of the main difficulties to create a general source-to-source translator is that Perl has a Turing-complete grammar in such a way that parsing can be affected by run-time code executed during the compile phase. Therefore, Perl cannot be parsed by a straight Lex/Yacc lexer/parser combination. Instead, the interpreter implements its own lexer, which coordinates with a modified GNU Bison parser to resolve ambiguities in the language [5], [6].

Some efforts have been done to integrate (not to translate) Perl into Java code. This is the case of the Java-Perl Library (JPL) [7], which allows to invoke Perl methods inside a Java program. We discard this solution because the performance obtained by JPL codes is equivalent to the one obtained by using directly Perl codes and Hadoop Streaming.

On the other hand, we cannot forget that Perl and Java are two very different languages. There are a lot of differences between them, but the following are the most relevant to our case:

- **Variable declaration:** Programmers do not have to declare and establish the variable type in Perl, while that is mandatory in Java.
- **Array size and accesses:** If the programmers want to access a non existent array position in Perl, the array is expanded if it is a write operation, and positions in the middle are set to *undef*, or, in the case of reading, it returns *undef* [8]. However, Java produces an execution error.
- **Boolean values:** Perl does not have boolean values such as "True" and "False", which can be assigned to variables. Instead, it can handle another variables as booleans. For example, the strings "" and "0" are considered as "False", and also the integer and real values 0 and 0.0 [8]. Java uses boolean variables, and it cannot process integers, real or strings as booleans like Perl.

Due to all the aforementioned difficulties, our objective in this work was not to develop a powerful tool that allows to automatically translate any existent Perl code to Java, but a simple and easy-to-use tool that takes as input Perl codes written for Hadoop Streaming, following a reduced number of additional programming rules, and produces Hadoop-ready Java codes. We have called this tool Perldoop, and it was developed using Python.

A. How it works

Perldoop uses file templates and tags, as it is shown in Figure 1. The main goal of the templates is to contain certain parts of the Java code that has no direct translation from Perl, such as class declarations, some auxiliary functions needed in Java or global variables. In the near future, and as Perldoop evolves, we expect that templates will not be necessary.

The programmer must indicate the position to insert the translated code into the file template using the next tags:

`//<java><start>` and `//<java><end>`

In the same way, the Perl code to be translated needs to be surrounded by the following tags:

`#<perl><start>` and `#<perl><end>`

The translation process can be summarized in these steps:

1) The programmer tags the Perl file and creates the Java template, including the class declaration and constructor.

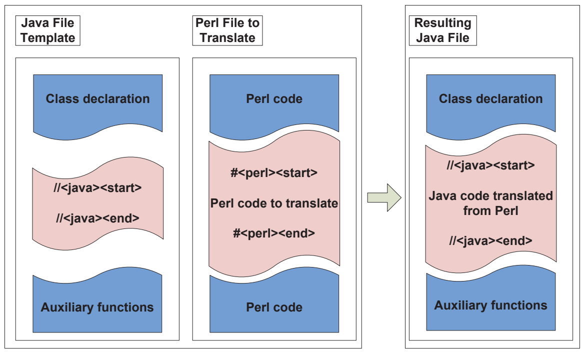


Fig. 1. Use of templates and tags with Perldoop.

2) The Perldoop tool is executed to generate the new Java code.

The main benefit of using this methodology is the simplicity of use. Note that programmers have to insert the labels in the Perl code and create the templates only once. After that procedure, the Perl code to be translated can be modified at any time. To obtain a new Java version of the code it is only necessary to execute Perldoop again, and it will be automatically generated.

Regarding the limitations of Perldoop, the main one is that programmers have to tag the Perl code and make the corresponding Java template. In addition, there are a few tips or rules that the programmer should follow in order to guarantee the correct translation when using Perldoop.

B. Programming rules

Next we detail the programming rules that the Perl codes should follow to assure the correctness of the Java codes automatically generated by Perldoop:

1) Ordered conditional blocks. It means that the conditional expression should appear before the sentences to be executed if the condition is fulfilled. Use:
```
if(condition){
  sentences;
}
```
instead of: `sentences if(condition);`
2) Perform string concatenations with the "." operator. For example:
```
$variable = $var1." ".$var2;
```
3) Restrict the access to array positions not previously allocated.
4) Use a different name for each variable. Perl allows:
```
my $feat; # variable
my @feat; # array
```
while in Java the programmer has to declare two different variables.
5) In Perl, the programmer can do the following using an integer variable: `if($variable)`
Java only allows this expression if the variable is a boolean. Therefore, the expression should be:
```
if($variable!=0)
```
A similar situation arises when using hashes in Perl.

```
#!/usr/bin/perl -w

#<perl><start>

my $line;               #<var><string>
my @words;              #<array><string>
my $key;                #<var><string>
my $valueNum = "1";     #<var><string>
my $val;                #<var><string>

while ($line = <STDIN>) {       #<map>
  chomp ($line);
  @words = split ("␣",$line);
  foreach my $w (@words) {      #<var><string>
    $key = $w."\t";
    $val = $valueNum."\n";
    print $key.$val;
  }
}

#<perl><end>
```

```
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public static class WordCountMap extends Mapper<Object, Text, Text, Text>{
  @Override
  public void map(Object incomingKey, Text value,
  Context context) throws IOException, InterruptedException {
    try{

      //<java><start>
      String line;
      String[] words;
      String key;
      String valueNum  = "1";
      String val;
      line   = value.toString();
      line = line.trim();
      words  = line.split("␣");
      for (String w : words) {
        key  = w+;
        val  = valueNum;
        context.write(new Text(key),new Text(val));
      }
      //<java><end>

    }
    catch(Exception e){
      System.out.println(e.toString());
    }
  }
}
```

Fig. 2. WordCount mapper example using Perl (top) and its equivalent Java code generated using Perldoop (bottom).

6) Declare and initialize the variables with the corresponding label, which includes the data type and class (variable, array, etc.). For example:

   `#<var><string>`
   `#<array><integer>`
   `#<arraylist><string>`

   In addition, we must take into account that Boolean values are not available in Perl. The next label is used to identify a boolean variable:

   `#<var><boolean>`

Additionally, programmers should also include a label to indicate if the Perl code corresponds to a mapper (`<map>`) or a reducer (`<reduce>`).

As we have commented previously, Perl is well-known for its unrivaled ability to process text using very powerful features such as regular expressions. The native Java support for regular expressions is not as good as the provided by Perl. A list of the main differences can be found in [9]. For this reason, in order to improve the handle of regular expressions in Java, Perldoop takes advantage of the *jregex* library [10]. This library uses Perl 5.6 regex syntax, including lookahead/lookbehind assertions and it holds a BSD license.

C. Example: WordCount in Perl

Next we present, for illustrating purposes, an example of the use of Perldoop. The goal is to translate a simple Perl

```
#!/usr/bin/perl -w

#<perl><start>

my $count = 0;          #<var><integer>
my $value;              #<var><integer>
my $newkey;             #<var><string>
my $oldkey;             #<var><string><null>
my $line;               #<var><string>
my @keyValue;           #<var><string>

while ($line = <STDIN>) {       #<reduce>
  chomp ($line);
  $keyValue = split ("\t",$line);

  $newkey = $keyValue[0];
  $value  = $keyValue[1];

  if (!(defined($oldkey))) {
    $oldkey = $newkey;
    $count  = $value;
  }
  else {
    if ($oldkey eq $newkey) {
      $count = $count + $value;
    }
    else {
      my $returnKey   = $oldkey."\t";   #<var><string>
      my $returnValue = $count."\n";    #<var><string>
      print $returnKey.$returnValue;
      $oldkey = $newkey;
      $count  = $value;
    }
  }
}
my $returnKey   = $oldkey."\t";         #<var><string>
my $returnValue = $count."\n";          #<var><string>
print $returnKey.$returnValue;

#<perl><end>
```

```
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public static class WordCountReducer extends Reducer<Text, Text, Text, Text> {
  int count  = 0;
  @Override
  public void reduce(Text key, Iterable<Text> values,
  Context context) throws IOException, InterruptedException {

    //<java><start>
    int value;
    String newkey;
    String oldkey = null;
    String line;
    for (Text val : values) {
      newkey    = key.toString();
      value     = Integer.parseInt(val.toString());
      if (!(oldkey!= null)) {
        oldkey   = newkey;
        count    = value;
      }
      else{
        if (oldkey.equals(newkey) ) {
          count   = count + value;
        }
        else{
          String returnKey    = oldkey;
          String returnValue  = String.valueOf(count);
          context.write(new Text(returnKey),
          new Text(returnValue));
          oldkey   = newkey;
          count    = value;
        }
      }
    }
    String returnKey    = oldkey;
    String returnValue  = String.valueOf(count);
    context.write(new Text(returnKey),new Text(returnValue));
    //<java><end>

  }
}
```

Fig. 3. WordCount reducer example using Perl (top) and its equivalent Java code generated using Perldoop (bottom).

script into a Hadoop-ready Java class. In particular, we have selected the Perl version of the WordCount, which counts the number of occurrences of each word in a text document. This code is a typical example used to test a Hadoop infrastructure.

Figures 2 and 3 show, on the top, the Perl code of the WordCount mapper and reducer, respectively. Note that these codes have been tagged following the programming tips detailed above. Next, the programmer has to create the Java templates with the class declaration where the translated codes will be inserted. Note that templates correspond to the Java code not included between `<java><start>` and `<java><end>`. After this step, Perldoop is executed. The resulting Hadoop-ready Java codes are displayed on the bottom of Figures 2 and 3. We must highlight that Perldoop can be applied to any Perl code ready to be executed with Hadoop Streaming.

III. Case studies: NLP scripts

Natural Language Processing (NLP) is considered as one of the methodologies more suited to structure and organize the textual information accessible through Internet. Linguistic processing of large amount of text is a complex task that requires the use of several subtasks organized in interconnected modules. Most of the existent NLP modules are programmed using Perl due to its unique ability to process text using regular expressions. One of the main problems found by the researchers in the NLP area is the high computational cost of their tools, which makes them impractical for the analysis of big volumes (Gigabytes or even Terabytes) of documents. As a consequence, the use of parallelism and Big Data technologies is mandatory in order to overcome these limitations.

Perldoop has been tested using several NLP modules. In particular, we have used a set of linguistic modules [11], [12]. These modules are written in Perl, and perform accurate linguistic annotation on large amounts of text corpora. The whole architecture consists of a pipeline in which these modules are chained, in such a way that the output of each module feeds directly the input of the next one. The text is linguistically annotated at increasingly complex level of analysis, i.e. sentence segmentation, tokenization, word splitting, Named Entity Recognition (NER), Part-of-Speech (PoS) tagging, and Named Entity Classification (NEC). The analysis chain has more than 150 regular expressions. In this section, we will briefly describe the last three processes in the analysis chain: NER, PoS-tagging, and NEC. These modules have been integrated into a Hadoop infrastructure using Perldoop. Note that these modules are suited to be used in more complex and higher level linguistic applications such as machine translation, information retrieval, question answering, or even new intelligent systems for technological surveillance and monitoring.

- *Named Entity Recognition (NER)*: This task consists of identifying as a single unit (or token) those words or chains of words denoting an entity, e.g. *New York*, *University of San Diego*, *Herbert von Karajan*, etc. The module is based on a set of language-independent rules that take into account information on both a large lexicon of forms and the relative position of words within the sentence.
- *PoS-Tagging*: This module assigns each token of the input text a single PoS tag provided with morphological information e.g. *singular and masculine adjective*, *past participle verb*, *plural and feminine noun*, etc. The module consists of a Bayesian classifier whose features are bigrams of tokens which represent the immediate left and right contexts of the target token [11][13]. The module makes use of the same tagset and lexicon as FreeLing, a well-known suite of multilingual linguistic tools [14].
- *Named Entity Classification (NEC)*: The last step of the linguistic analysis is the semantic classification of those entities identified in the previous NER step. Named Entity Classification (NEC) is the process of classifying entities by means of classes such as “People”, “Organizations”, “Locations”, or “Miscellaneous”. NEC is a crucial task for several natural language applications, namely Question Answering and Information Extraction. The NEC module relies on a distant-supervised strategy and consists of two tasks. First, large resources (e.g. gazetteers of persons, locations, and organizations) are automatically generated with the aid of encyclopedic data stored in databases such as FreeBase [15] and DBpedia [16]. Second, a set of disambiguation rules are applied on previously identified entities, in order to solve both ambiguous and unknown entities. This module was described in [12] and reached state-of-the-art precision on different test corpora.

IV. Performance evaluation

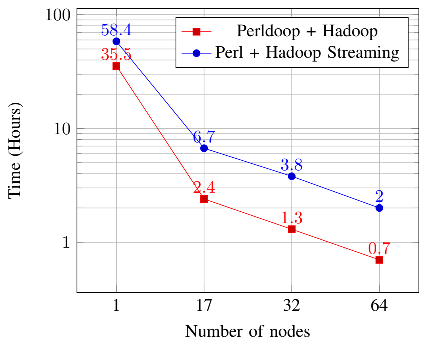
The experiments shown in this section were performed on a Hadoop cluster installed at the Galicia Supercomputing Center (CESGA), which consists of 64 nodes. Each node has an Intel Xeon E5520 processor and 1 GB of RAM memory. The Hadoop version is the 1.1.2, while the Java and Perl versions are 1.7.0 and 5.10.1 respectively. The performance results for the NLP Perl modules considered in the work (NER, Tagger and NEC) were obtained using the Wikipedia in plain text (file size of 2.1 GB) as input. The block size was 128 MB.

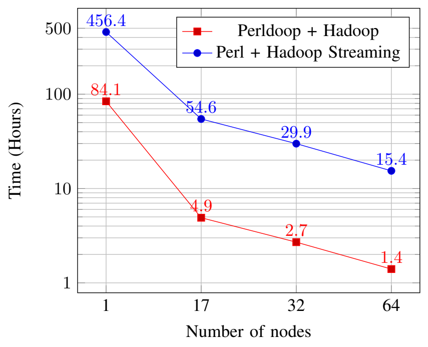
As a first approach, we have integrated the sequential NLP Perl modules into a Hadoop infrastructure using the Hadoop Streaming tool. Afterwards, the same Perl codes were automatically converted into Hadoop-ready Java codes using the Perldoop tool. A performance comparison of both approaches is shown next. Note that in this particular case only mappers were generated because reducers are not necessary.

Figures 4 shows the execution times of the NER, Tagger and NEC modules on the cluster using both, Hadoop Streaming and the new automatically generated Java codes with Hadoop (Perldoop + Hadoop in the figures). Different number of nodes were considered. Note that 17 nodes were used instead of 16 because, for the latter, the split size was bigger than the block size. An important reduction of the processing time is observed for all the parallel executions with respect to the sequential case, both using Hadoop Streaming and Hadoop. For example, the original Tagger and NEC modules require about 19 days (more than 450 hours) to process the whole Wikipedia, while using Hadoop Streaming this time reduces to less than 16 hours using 64 nodes. Despite these important improvements, the execution times using Hadoop Streaming are still too high.

Java codes generated by Perldoop using one node behave better than their Perl counterparts, but the processing times are also very high. Considering the parallel executions, the performance of the new Java modules clearly outperforms the Perl ones, reducing the processing times to less than 2 hours for all the NLP modules when using 64 nodes.

Figure 5 shows the speedups obtained by the Perldoop generated Java modules using Hadoop with respect to the original Perl codes using Hadoop Streaming. The performance gains range from $1.6\times$ to $12.1\times$. For parallel executions, we





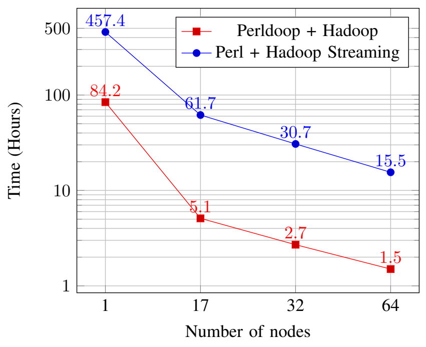


Fig. 4. Execution time of the NER (left), Tagger (center) and NEC (right) modules on a Hadoop cluster (log scale).

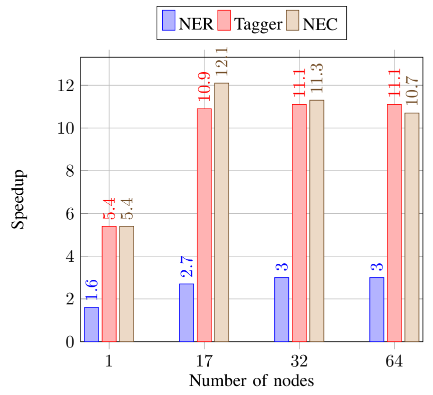


Fig. 5. Performance improvement of the Java modules generated by Perldoop using Hadoop with respect to the use of Perl and Hadoop Streaming.

must highlight that the Tagger and NEC modules process the Wikipedia, at least, 10× faster than using Hadoop Streaming. The worst behavior is observed for the NER module, which is the NLP module with low computational cost. Despite of that, speedups up to 3× are reached. These results confirm the important performance differences between using Java codes with Hadoop, and the Perl modules with Hadoop Streaming.

But, in addition to decrease the processing times, the scalability of the new NLP modules also improves when using Hadoop. Figure 6 shows the speedup of the Java and Perl versions of the modules when using Hadoop and Hadoop Streaming. While the speedups observed with Hadoop Streaming are far from the ideal case, the new Java modules generated by Perldoop are closer to it. For example, considering 64 nodes, the speedups of the NER, Tagger and NEC modules are 51.2×, 60.8× and 57.9× respectively.

V. Related work

With respect to the Hadoop performance, several studies have compared Hadoop Streaming with pure Hadoop Java codes and probed that Hadoop Streaming degrades the performance a lot for data intensive jobs [4]. Other authors proposed improvements to the Hadoop Streaming code [26] or developed their own MapReduce framework on Hadoop [27]. Alternatively, in [28] a Python based programming model for MapReduce similar to the Java one is presented, which provides a Python API for both the MapReduce and the distributed file system using Hadoop Pipes. In any case, despite the improvement gained over Hadoop Streaming, Java codes still have a better performance in Hadoop.

In the last few years, some work has been carried out to use Big Data technologies (mainly the MapReduce programming models) to deal with some aspect of NLP. In statistical translation, MapReduce has been used in [17] and [18]. In [19], the author uses Hadoop to build word co-occurrence matrices from large corpora, whilst Pantel et al. [20] use the MapReduce framework for computing the pairwise semantic similarity between words and in [21] the authors use it for paraphrase acquisition. Most of these works developed ad-hoc solutions adapted to the MapReduce paradigm, using Java in order to get the best performance.

Other authors proposed to adapt existing codes, written in scripting languages like Python, to the MapReduce framework. So, in [22], Hadoop Pipes and SWIG [23] have been used to integrate NLTK (*Natural Language Toolkit*) [24], which is written in Python, into Hadoop. Hadoop Pipes provides slightly better performance than Streaming, but it is worse than Java. On the other hand, Attardi et al. [25], present a suite of tools for text analytics based on the software architecture paradigm of data pipelines, using a modified version of Hadoop Streaming that allows them to have an ordered output. Unlike those works, the solution presented in this paper uses previously developed Perl codes, which effortlessly are translated into Java code ready to be executed in Hadoop. So, it combines the expressiveness and power of Perl regular expressions with the good performance of Java codes running in Hadoop.

VI. Conclusions

Hadoop is the most important implementation of the MapReduce programming model. It provides an utility to execute applications written in languages different from Java, known as Hadoop Streaming. However, the ease of use provided by Hadoop Streaming comes at the expense of noticeable degradations in the performance.

In this work, we introduce Perldoop, a new tool that automatically translates Hadoop Streaming scripts written in Perl into Hadoop-ready Java codes. Perldoop is a simple and easy-to-use tool that takes as input Perl codes written following a reduced number of programming rules, and produce Hadoop-ready Java codes. To the best of our knowledge, this is the first tool to deal with this problem.

Perl is well-known for its unrivaled ability to process text using very powerful features like regular expressions. For this reason, a lot of Natural Language Processing (NLP) applications have been developed using this language. We have tested

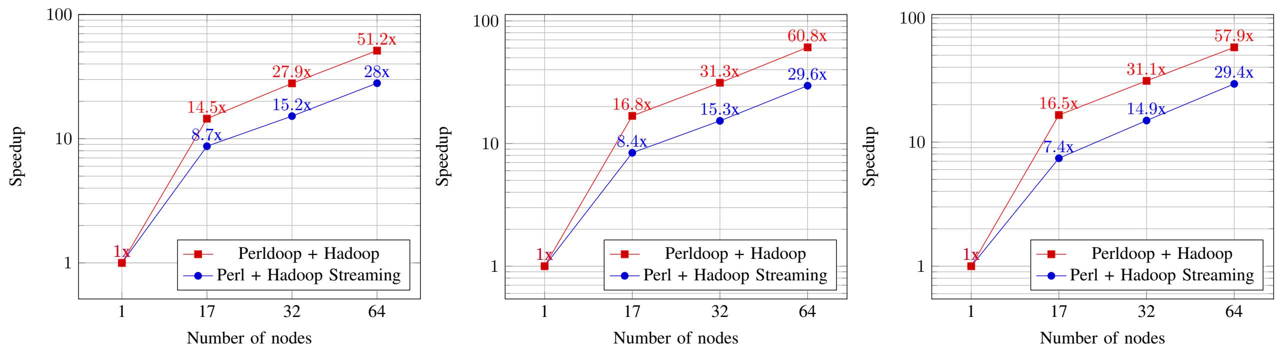


Fig. 6. Speedup with respect to the sequential version in Java and Perl for the NER (left), Tagger (center) and NEC (right) modules (log scale).

our tool using several NLP modules that perform accurate linguistic annotation on large amounts of text corpora. In particular, the linguistic modules considered in this work carry out the following tasks: Named Entity Recognition (NER), Part-of-Speech (PoS) tagging, and Named Entity Classification (NEC). These modules were automatically translated into Hadoop-ready Java codes using Perldoop.

A performance comparison using the original Perl scripts with Hadoop Streaming, and the new Java codes with Hadoop was performed on a cluster. An important decrease in the processing times was observed with respect to the sequential case, both using Hadoop Streaming and Hadoop. However, the performance of the new Java modules clearly outperforms the Perl ones, reaching speedups up to 12×. We must highlight that the new Java modules reduce the time required to process the whole Spanish Wikipedia to less than 2 hours when using 64 nodes, demonstrating the benefits of using Perldoop.

Acknowledgments

This work has been supported by the Xunta de Galicia (Spain) grant EM2013/041. The authors also wish to thank CESGA for providing access to their supercomputing facilities.

References

[1] IBM, "Big data at the speed of business," http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html, [Online; accessed July, 2014].

[2] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] Apache Hadoop, http://hadoop.apache.org, [Online; accessed July, 2014].

[4] M. Ding, L. Zheng, Y. Lu, L. Li, S. Guo, and M. Guo, "More convenient more overhead: the performance evaluation of Hadoop streaming," in *ACM Symp. on Research in Applied Computation*, 2011, pp. 307–313.

[5] J. Kegler, "Perl and undecidability: The halting problem," *The Perl Review*, vol. 4, pp. 21–25, 2008.

[6] ——, "Perl and undecidability: Perl is undecidable," *The Perl Review*, vol. 5, pp. 7–11, 2008.

[7] L. Wall, B. Jepson, N. Patwardhan, E. Siever, and D. Futato, *PERL Resource Kit UNIX Edition: 4 Volume Set with CD-ROM*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1997.

[8] "Perl programming documentation," http://perldoc.perl.org/, [Online; accessed July, 2014].

[9] Oracle, "Java platform, standard edition 7 API specification," http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#jcc, [Online; accessed July, 2014].

[10] JRegex, Regular Expressions for Java, http://jregex.sourceforge.net/, [Online; accessed July, 2014].

[11] P. Gamallo and M. García, "Using morphosyntactic post-processing to improve PoS-tagging accuracy," in *9th Int. Conf. on Computational Processing of Portuguese Language (PROPOR)*, 2010.

[12] ——, "A resource-based method for named entity extraction and classification," *LNCS series*, vol. 7026, pp. 610–623, 2011.

[13] M. Banko and R. Moore, "Part of speech tagging in context," in *Proc. of the 20th Int. Conf. on Computational Linguistics*, 2004.

[14] L. Padró and E. Stanilovsky, "Freeling 3.0: Towards wider multilinguality," in *Proc. of the LREC*, 2012.

[15] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *Proc. of the ACM SIGMOD Int. Conf. on Management of data*, 2008, pp. 1247–1250.

[16] J. Lehman et al., "DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia," *Semantic Web Journal*, 2014.

[17] C. Dyer, A. Cordova, A. Mont, and J. Lin, "Fast, easy, and cheap: Construction of statistical machine translation models with MapReduce," in *Proc. of the Workshop on Statistical Machine Translation*, 2008, pp. 199–207.

[18] R. Ahmad, P. Kumar, B. Rambabu, P. Sajja, M. K. Sinha, and R. Sangal, "Enhancing throughput of a machine translation system using MapReduce Framework: An engineering approach," in *9th Int. Conf. on Natural Language Processing*, 2011.

[19] J. Lin, "Scalable language processing algorithms for the masses: A case study in computing word co-occurrence matrices with MapReduce," in *Proc. of the EMNLP*, 2008, pp. 419–428.

[20] P. Pantel, E. Crestan, A. Borkovsky, A.-M. Popescu, and V. Vyas, "Web-scale distributional similarity and entity set expansion," in *Proc. of the EMNLP*, 2009, pp. 938–947.

[21] D. Metzler and E. Hovy, "Mavuno: a scalable and effective Hadoop-based paraphrase acquisition system," in *Proc. of the 3rd Workshop on Large Scale Data Mining: Theory and Applications*, 2011, p. 3.

[22] P. Bone, "Integrating NLTK with the Hadoop MapReduce framework – 433-460 Human Language Technology Project," 2008.

[23] D. M. Beazley, "SWIG: An easy to use tool for integrating scripting languages with C and C++," in *Proc. of the 4th USENIX Tcl/Tk workshop*, 1996, pp. 129–139.

[24] S. Bird, "NLTK: the natural language toolkit," in *Proc. of the COLING/ACL on Interactive presentation sessions*, 2006, pp. 69–72.

[25] G. Attardi, S. D. Rossi, and M. Simi, "The Tanl pipeline," in *Proc. of the 7th Int. Conf. on Language Resources and Evaluation*, may 2010.

[26] L. Lai et al., "ShmStreaming: A shared memory approach for improving Hadoop streaming performance," *Int. Conf. on Advanced Information Networking and Applications*, pp. 137–144, 2013.

[27] E. Dede et al., "MARISSA: MApReduce implementation for streaming science applications." in *eScience*, 2012, pp. 1–8.

[28] S. Leo and G. Zanetti, "Pydoop: a Python MapReduce and HDFS API for Hadoop," in *Proc. of the 19th ACM Int. Symposium on High Performance Distributed Computing*, 2010, pp. 819–825.